Master Thesis

Abstract Rewriting Formalized in Lean

From Foundations to Completeness of 2-label DCR for Countable Systems

Sam van Kampen (2665265)

Supervisors:
Femke van Raamsdonk
Jörg Endrullis

Second reader:
Kristina Sojakova

A thesis submitted in partial fulfillment of the requirements for the joint UvA-VU Master of Science degree in Computer Science

January 6, 2025

Contents

1 Introduction

Although they may never realize it, most people come into contact with rewriting at an early age. At primary school, we learn about the natural numbers, along with the basic arithmetic operations, and we learn to simplify arithmetic expressions, by rewriting subterms step-by-step until we can no longer apply any rewrite rules.

$$\begin{tikzcd}
 & 5 \cdot (3 + 1) \arrow[dl] \arrow[ddr] & \\
5 \cdot 3 + 5 \cdot 1 \arrow[dd] & & \\
 & & 5 \cdot 4 \arrow[ddl] \\
15 + 5 \arrow[dr] & & \\
 & 20 &
\end{tikzcd}$$

Figure 1: When simplifying arithmetic expressions, any diverging steps will converge again; the rewriting system we use to simplify arithmetic expressions is *confluent*.

The field of *rewriting* is concerned with all systems that are characterized by this step-by-step application of rules which transform an object into another. Rewriting appears in many forms, but it is most well-known in the form of *term rewriting*, which was largely developed in the 1930s as the basis for the lambda calculus, and still forms the basis for modern functional programming languages. In this thesis, we will look at *abstract rewriting*, the subfield of rewriting that looks at rewriting systems in their most general form, as a collection of binary rewrite relations on an unspecified set of objects.

Despite knowing nothing about our objects, we can still define various important properties of our rewrite relation. For instance, consider the rewriting steps shown in Fig. 1. Although we take diverging steps from our initial expression $5 \cdot (3 + 1)$ to the intermediate expressions $5 \cdot 3 + 5 \cdot 1$ and $5 \cdot 4$, these intermediate expressions eventually converge again in our final simplified expression 20. This property that diverging steps always converge again is called *confluence*. Additionally, it is impossible for us to keep simplifying an arithmetic expression forever; regardless of our strategy, any rewriting process will eventually terminate, resulting in a *normal form*, an expression that cannot be simplified any further. This property is called *strong normalization*, or alternatively, *termination*.

Confluence and termination are generally desirable properties to have, since they guarantee that applying the rules in our rewriting system will always produce a final result, which is the same regardless of the order in which we apply the rewrite rules.

Unfortunately, both confluence and termination are undecidable – that is, there is no universal procedure for determining whether an arbitrary rewriting system is confluent or terminating. In fact, it is often difficult to directly prove that a rewriting system is confluent. Instead, we often show a rewriting system is confluent by showing that it satisfies various simpler properties, which together imply confluence. Over the years, many of these so-called *confluence criteria* have been developed: the Hindley-Rosen lemma, Rosen's requests lemma, Newman's lemma, et cetera.

One of the reasons for this wealth of confluence criteria is that they are generally incomplete. That is, if a system does not satisfy a confluence criterion, it does not necessarily mean that it is not confluent; there might be other confluence criteria that it *does* satisfy. That said, there are some results that show that confluence criteria are complete on a specific class of rewriting systems.

In this thesis, we are concerned with formalizing much of the basic theory of abstract rewriting, culminating in a proof that one specific confluence criterion, *2-label decreasing diagrams*, is complete for the class of countable rewriting systems.

1.1 Contributions

Our main contribution is a formalization of Klop, Endrullis and Overbeek's proof of the completeness of 2-label DCR for countable systems. Additionally, we have formalized many of the foundational notions and results in abstract rewriting, both where necessary to complete our main formalization and in an attempt to construct a more or less complete formalized foundation of abstract rewriting theory. This includes basic properties on rewrite relations (confluence, normalization, ...), as well as various lemmas interrelating these properties, chief among them three proofs of Newman's Lemma. Figure 2 gives an overview of these proofs, inspired by a similar figure in [8].

Our formalization is available as a Lean project in the Git repository at `https://github.com/svkampen/msc-thesis/`.

1.2 On formalization

With this thesis, we are making a small contribution to the collection of formalized mathematics. We think this is worthwhile for multiple reasons.

- **Formalization provides strong evidence.** Although all forms of proof have pitfalls, formal proof included, a formal proof is relatively strong evidence that a result holds, since it often makes explicit many implicit assumptions and steps in an informal proof and requires us to justify every minor proof step, relying only on a small collection of axioms. This is the traditional selling point of formalization.
- **Formalization cements understanding.** Because formalization makes explicit every implicit assumption and proof step in an informal proof, it makes the formalizer con-

$$\begin{tikzcd}
 & \mathit{CR}^{\leq 1} \arrow[d] & & \\
\mathit{NF} \arrow[d] & \mathit{CR} \arrow[l] \arrow[r, shift left] \arrow[dd, shift left] & \mathit{CP} \arrow[l, shift left, "countable"] \arrow[d] \arrow[dr] & \\
\mathit{UN} \wedge \mathit{WN} \arrow[ur] \arrow[dd] & & \mathit{DCR} & \mathit{DCR}_2 \arrow[l] \\
 & \mathit{SN} \wedge \mathit{WCR} \arrow[uu, shift left] \arrow[ul] & & \\
\mathit{Ind} \wedge \mathit{Inc} \arrow[ur] & \boxed{\mathit{WCR} \wedge \mathit{WN} \wedge \mathit{Inc}} \arrow[u] & &
\end{tikzcd}$$

Figure 2: An overview of the interrelations between ARS properties that we have formalized. Note that there are implications which are true, but not shown in this figure (such as DCR $\Rightarrow$ CR), because they are not formalized in this work. Arrows pointing from or to a conjunction symbol refer to the entire conjunction; pointing from or to a conjunct refer only to that conjunct.

tend with the gaps or misunderstandings in their own knowledge of the subject. On multiple occasions, while in the middle of writing a Lean proof, I have realized that my own understanding of a proof step or definition was incomplete or subtly wrong, and my understanding of the subject has gotten better for it.

- **Formalization is fun.** Formalization using an interactive theorem prover has been compared to a video game multiple times. By turning the solo activity of writing a proof into an interactive activity where you 'play' against a computer which tries to refute your arguments, theorem proving becomes more like a puzzle game, and although it can sometimes be extremely frustrating, it is mostly a lot of fun.

Our theorem prover of choice in this work is Lean. One might wonder: why Lean? A number of results in abstract rewriting have already been formalized in Isabelle/HOL, so that seems like a natural choice here, as well. And more generally, there is a wealth of proof assistants to choose from these days. In large part this is simply personal preference: I was taught Lean as a student and am therefore well-versed in it. Additionally, because very little rewriting has been formalized in Lean, we have the opportunity to build up the theory of abstract rewriting from the ground up, trying out different Lean representations of definitions and seeing which ones work best in order to prove our results, without necessarily being influenced by the decisions made by earlier formalization work.

1.3 Related work

Abstract rewriting is generally treated as a precursor to more concrete areas of rewriting, in particular term rewriting, and as such, the literature on abstract rewriting is often found as introductory chapters in books on term rewriting. The aptly named *Term Rewriting Systems* [8], which treats abstract rewriting in chapters 1 and 14, is the basis for much of this thesis.

A number of results in abstract rewriting have been previously formalized by Thiemann and Sternagel in Isabelle/HOL as part of the IsaFoR/CeTA project [7, 9]. Their Isabelle theory has also been used in various other results, among them Harald Zankl's formalization of Confluence by Decreasing Diagrams [11]. Various elements of this formalization are inspired by their work.

1.4 Terminology

In many rewriting systems, transforming an object by means of a rewrite rule in some way 'reduces' the object – for instance, in Fig. 1, our initial complex expression $5 \cdot (3 + 1)$ is reduced to the normal form 20. For this reason, the term rewrite is often replaced by the term reduction, e.g. *abstract reduction system*, *reduction relation*, *reduction sequence*. We use the terms interchangeably.

1.5 Lean conventions

The rest of this thesis contains numerous snippets of Lean code. The following variables may be assumed to exist in any Lean code snippet included from chapter 3 onwards:

```
variable (α I: Type*) (r s: Rel α α) (A: ARS α I)
```

Note that we use the standard Lean convention of denoting arbitrary type variables by a Greek letter (`α, β, γ, ...`). This occasionally clashes with the use of Greek letters for indices in the literature. We instead use the letters $i, j, k, \ldots$ for these indices.

When describing definitions and lemmas for the first time, we will often link to their page in the library documentation. These links are printed in `Monospaced Green`, e.g. `Set`.

1.6 Acknowledgements

I would like to thank Edward van de Meent and Daniel Weber from the Lean Zulip for their contributions to formalizing the expansion of (reflexive-)transitive reduction sequences.

2 Lean

Lean is an interactive theorem prover based on dependent type theory – specifically, the *Calculus of Inductive Constructions* (CIC). CIC is also used in other theorem provers, chief among them Coq, from which it originates. Dependent type theory has a computational interpretation, and as such, Lean can be used as a functional programming language. For instance, consider the following snippet of Haskell code, which defines the datatype `Nat` of natural numbers, as well as an addition function on `Nat`:

```
data Nat where
  Zero :: Nat
  Succ :: Nat -> Nat

add :: Nat -> Nat -> Nat
add Zero b     = b
add (Succ a) b = Succ (add a b)
```

The equivalent Lean code looks fairly similar, using an inductive type for `Nat` and similarly performing pattern matching in `add`:

```
inductive Nat where
  | zero: Nat
  | succ: Nat → Nat

def add: Nat → Nat → Nat
  | Nat.zero, b      ⇒ b
  | (Nat.succ a), b ⇒ Nat.succ (add a b)
```

Since Lean is primarily a research project, it has undergone significant changes over the past few years. That said, at this time, the migration from Lean 3 to Lean 4 is more or less completed, and there are currently no big changes to the language planned. Whereas previous versions of Lean were intended to be theorem provers first, with the ability to encode and evaluate programs second, Lean 4 forms an integrated functional programming language and theorem prover, similar in some sense to Agda.

As such, Lean has all the features you would expect from a modern functional programming language. What makes Lean special is its support for proving properties of our definitions. Alongside its computational interpretation, dependent type theory also has a logical interpretation. Namely, we can state propositions as types, and give proofs as inhabitants of that type. Take, for example, logical implication. The statement $p \implies q$ in propositional logic can be seen as the function type `p → q`: if we have a function of type `p → q`, and we provide an element of type `p`, we can get an element of type `q`, just as, by modus ponens, a proof of p and a proof of $p \implies q$ yield a proof of q. Therefore, if we can define a function `p → q`, we have a proof of $p \implies q$.

Of course, implication is only one of the standard logical connectives. Lean's type theory also allows us to encode the rest of the logical primitives of higher-order logic. For instance, conjunction and disjunction (`And` and `Or`):

```
inductive And: Prop → Prop → Prop where
  | intro: (a b: Prop) → a → b → And a b

inductive Or: Prop → Prop → Prop where
  | inl: (a b: Prop) → a → Or a b
  | inr: (a b: Prop) → b → Or a b
```

Here, we define the inductive types `And` and `Or`. Both *type constructors* `And` and `Or` take two propositions and themselves represent propositions; this is represented in their type signature `Prop → Prop → Prop`. `And` has a single introduction rule, which takes two propositions `a`, `b` as well as proofs of `a` and `b`, and returns a proof of `And a b`, more commonly written `a ∧ b`. `Or` instead has two introduction rules, because there are two ways of constructing a proof of $a \lor b$: either by showing a is true, or by showing b is true. Therefore, we have one introduction rule which takes a proof for the left disjunct, and one which takes a proof for the right disjunct.

The perceptive reader may notice some unfamiliar syntax in the introduction rules for `And` and `Or`. Instead of a regular function type, which takes the form `σ → τ`, we are greeted by the dependent function type `(x: σ) → τ[x]`, where `x` has the type `σ`, and may appear in `τ`. This allows `τ` to *depend* on `x` – hence the name dependent type theory.

Although the syntax above shows the underlying dependent function types well, we generally use slightly different syntax, which moves the common parameters to the header. Additionally, since `And` is a single-constructor type, we use the `structure` keyword, which allows us to refer to the left and right conjunct using the `And.left` and `And.right` functions (also written `h.left` and `h.right` if `h: a ∧ b`). The following definitions come straight from the Lean standard library (`Init/Prelude.lean`):

```
structure And (a b : Prop) : Prop where
  /-- `And.intro : a → b → a ∧ b` is the constructor for the And operation. -/
  intro ::
  /-- Extract the left conjunct from a conjunction. `h : a ∧ b` then
      `h.left`, also notated as `h.1`, is a proof of `a`. -/
  left : a
  /-- Extract the right conjunct from a conjunction. `h : a ∧ b` then
      `h.right`, also notated as `h.2`, is a proof of `b`. -/
  right : b

inductive Or (a b : Prop) : Prop where
```

```
/-- `Or.inl` is "left injection" into an `Or`.
    If `h : a` then `Or.inl h : a ∨ b`. -/
| inl (h : a) : Or a b
/-- `Or.inr` is "right injection" into an `Or`.
    If `h : b` then `Or.inr h : a ∨ b`. -/
| inr (h : b) : Or a b
```

Similar definitions exist for ⊤ (`True`), ⊥ (`False`), ∃ (`∃`, `Exists`), ⇔ (`↔`, `Iff`), and = (`=`, `Eq`). $\neg a$ (`¬`, `Not`) is defined as `a → False`, and ∀ is simply alternative notation for the dependent function type: `(∀x: σ, τ[x]) ↔ ((x : σ) → τ[x])`. This completes our complement of standard logical symbols.

Although proofs can be written in the same functional language as definitions and programs, most Lean proofs make use of the built-in *tactic language*. A similar system exists in Coq (*Ltac*), and there are loosely related concepts in Isabelle/HOL and Agda, although they work significantly differently. As an example, let's look at a proof about our `add` function defined above:

```
theorem add_zero: ∀a: Nat, add a 0 = a := by
  intro a            -- We introduce a, moving it into the context.
  induction a        -- We perform induction on a.

  /- The base case is trivial, by reduction of `add 0 0` to `0`. -/
  case zero ⇒
    trivial

  /- In the inductive step, we have `n: Nat, ih: add n 0 = n`,
     and we must show `add (n + 1) 0 = n + 1`. -/
  case succ n ih ⇒
    rw [add]         -- `add (n + 1) 0` = `(add n 0) + 1` (by `add`)
    rw [ih]          -- `(add n 0) + 1` = `n + 1` (by `ih`). QED.
```

Using the tactic language, the proof is written in a way that somewhat resembles pen-and-paper mathematics. Aside from basic operations (introduction, induction, cases, rewriting, ...), tactics are also the main way to access proof automation in Lean. For instance, there are tactics which automatically resolve integer and natural linear arithmetic problems (`omega`), simplify the goal or hypotheses in various ways (`simp`), et cetera.

Formalizing mathematics in Lean, then, is a question of defining types and functions, formulating propositions, and writing proofs. To help us get started, Lean ships with a number of built-in types, functions, and lemmas, which we can use in our formalization: the Lean *standard library*. Aside from the standard library, most of the mathematics that has been formalized in Lean lives in *mathlib*, the community-maintained mathematical library.

2.1 Set-theoretic definitions in Lean

Much of traditional mathematics is written in the language of set theory, and this is also the case with abstract rewriting. Since Lean is not based on set theory, but on type theory, we will often have to make some adjustments to set-theoretic definitions in order to represent them in Lean.

The differences are best illustrated using an example. Consider the following objects:

- $\mathbb{N}$, the natural numbers,
- $\{1, 2, 3\}$, a set of natural numbers,
- 1, a natural number.

In set theory, all three of these are sets, just like every other object in set theory. In this way, set theory is essentially untyped. This means we can state things like $\mathbb{N} = \{n \mid n \in \mathbb{N}\}$, or even $1 \in 2$, without violating the rules of set theory.

In Lean's type theory, these three objects also exist, but they are distinctly different kinds of object. The natural numbers are a *type*, with individual natural numbers (like 1) being inhabitants of the type. The only object that we would also call a set in type theory is our set of natural numbers $\{1, 2, 3\}$ – in Lean, this object would have the type `Set ℕ`. As you can see, a set in Lean is always a set of elements *of some type* – in this case, natural numbers. In some sense, the natural numbers are a 'base set' which would be represented as a type in Lean, and any subsets of a base set are what we call `Set`s in Lean.

Because these three kinds of objects have different types, many 'nonsensical' statements become invalid. For instance, ∈ is only defined if the right-hand side is a `Set` with elements with the same type as the left-hand side, so the expression `1 ∈ 2` is not well-typed. Additionally, trivial equalities in set theory like $\mathbb{N} = \{n \mid n \in \mathbb{N}\}$ are not well-typed in Lean, because the natural numbers are distinct from the set containing all natural numbers.

Aside from sets, Lean has the concept of *subtypes* (`Subtype`). Subtypes are essentially sets with the membership predicate lifted into the type system. That is, the set `s := { m: ℕ | m < 10 }` contains elements `n: ℕ` which satisfy `n < 10`, and the subtype `{ m // m < 10 }` has inhabitants `n : { m // m < 10 }`, which consist of a value `n.val : ℕ` and a proof `n.prop: n.val < 10`. Choosing between sets and subtypes is a trade-off: do you always want to carry the membership proof with the element? For instance, if you find yourself writing a lot of proofs that take an element of some type along with a proof that the element is a member of a set, it might be more convenient to use subtypes. Additionally, because subtypes are types, they can be used wherever a type is expected. Both subtypes and sets occur in various definitions in this thesis, and it is fairly easy to convert between the two.

3 Abstract Rewriting

As mentioned, unlike Isabelle/HOL, Lean does not yet contain any results in abstract rewriting. Therefore, we must build the theory of abstract rewriting from the ground up, starting with the foundational definitions and properties: reduction relations, abstract reduction systems, reduction sequences, confluence, and normalization.

3.1 Notation

The basic notation of abstract reduction systems and reduction relations is taken from [8, pp. 7–10]. We deviate from their notation occasionally, most notably when representing equivalence and equality. The notational conventions are included here for convenience.

Let A be a set. For a binary relation $r \subseteq A \times A$ we write $r^=$ for its reflexive closure, r^+ for its transitive closure, r^* for its reflexive-transitive closure, r^{-1} for its inverse, and $r^\equiv$ for its equivalence closure.

When the binary relation is represented as an arrow $\rightarrow$, we may additionally write $\twoheadrightarrow$ for its reflexive-transitive closure, $\leftarrow$ for its inverse, $\leftrightarrow$ for its symmetric closure and $\leftrightarrow^*$ for its equivalence closure. If $(a, b) \in \rightarrow$, we generally write $a \rightarrow b$.

If the relation in question is obvious, we might refer to two elements a, b being related by the equivalence closure simply as $a \equiv b$. The notation $a = b$ is reserved for true equality.

We will generally use $a, b, c, d, e, f, g, x, y, z$ to denote elements of some set $A, B, \ldots$; r, s, t to denote binary relations; i, j, k to denote indices; m, n to denote natural numbers; and $\mathcal{A}, \mathcal{B}, \ldots$ to denote abstract reduction systems.

3.2 Abstract reduction systems

In order to model the rewriting processes we have described in the introduction mathematically, we represent a rewrite rule as a binary relation $R \subseteq A \times A$ over our set of objects A, which contains a pair (a, b) if and only if a can be rewritten to b according to the rewrite rule. We call such a relation a rewrite (or reduction) relation.

Just as we may have many rewrite rules, we may have many reduction relations in one reduction system. These multiple reduction relations are generally modeled as a family of reduction relations, indexed by some set I.

Definition 3.1 (reduction, [8, p. 8]). Let A be a set of objects, I be a set of indices, and $\{\rightarrow_i \mid i \in I\}$ be a family of reduction relations, with every $\rightarrow_i \subseteq A \times A$.

(i) If $a \rightarrow_i b$, we call b a *one-step* (i-)*reduct* of a, and a a *one-step* (i-)*expansion* of b.

(ii) A *reduction sequence* with respect to $\rightarrow_i$ is a sequence $a_0 \rightarrow_i a_1 \rightarrow_i \cdots$, consisting of zero or more *reduction steps* $a_j \rightarrow_i a_{j+1}$. The sequence may be finite or infinite; if it is finite, it will have some final element b, which is called an (i-)*reduct* of a. We may also say *a reduces to b*.

(iii) The *length* of a finite reduction sequence is the number of reduction steps that it consists of.

(iv) An *indexed reduction sequence* is a finite or infinite sequence of indexed reduction steps, i.e. $a_0 \to_j a_1 \to_k a_2 \to_l \cdots$ with $i, j, k, \ldots \in I$.

Traditionally, the object set A and family of reduction relations I that make up a reduction system are bundled into a structure, which is called an *abstract reduction system*.

Definition 3.2. An *abstract reduction system* (ARS) is a structure $\mathcal{A} = (A, \{\to_i \mid i \in I\})$ consisting of a set A and a family of reduction relations $\to_i \subseteq A \times A$ indexed by some set I. If $(a, b) \in \to_i$, we write $a \to_i b$.

We refer to the union of reduction relations in an ARS as $\to_I = \bigcup_{i \in I} \to_i$, or simply $\to$. Two indexed ARSs $\mathcal{A} = (A, \{\to_i \mid i \in I\})$ and $\mathcal{B} = (A, \{\to_j \mid j \in J\})$ are *reduction-equivalent* if they have the same union of reduction relations, i.e. $\to_I = \to_J$.

Sources often distinguish between an *indexed* and *non-indexed* ARS, where a non-indexed ARS has only a single reduction relation. Since a non-indexed ARS is equivalent to an indexed ARS with a single index, we do not distinguish between the two.

As mentioned in Section 2.1, we will need to make some adjustments to these set-theoretic definitions to produce our Lean definitions. Instead of sets of objects and indices, our ARS structure takes a type `α` of objects and a type `I` of indices. To represent a rewrite relation, we could faithfully translate our set-theoretic definition $\to_i \subseteq A \times A$ as follows:

Definition 3.3 (A faithful translation of Definition 3.2 to Lean)**.**

```
structure ARS (α I: Type) where
  rel: I → Set (α × α)
```

In this definition, our family of rewrite relations is represented as a function, which takes an index and returns a rewrite relation, which is a subset of `α × α`. Although this is a faithful definition, it is not the way binary relations are generally represented in Lean, which is as a binary function `α → α → Prop`, also written `Rel α α`.

Let's consider how we can get from `Set (α × α)` to `α → α → Prop`. In Lean, a set is defined in terms of its membership predicate – in fact, the Lean type `Set β` is definitionally equal to `β → Prop`, the type of membership predicates[1]. That means the type of our rewrite relation can also be written `(α × α) → Prop`. Lastly, a function that takes a pair of elements can be *curried*, and written instead as `α → α → Prop`.

This is the standard form of a relation in Lean; using it in the form `Rel α α` allows us to, for instance, take the inverse of a relation `r` using `r.inv`, as well as use the *mathlib* definitions for the reflexive, transitive, reflexive-transitive and equivalence closure of a relation: `ReflGen`, `TransGen`, `ReflTransGen` and `EqvGen`. For example:

[1]Why does a membership predicate `m` have the type `β → Prop`? Because, for any element `b: β`, `m b` is the proposition which holds iff `b` is in the set.

Example 3.1 (Working with the reflexive-transitive closure of a relation).

```
/-- `is_succ a b` holds if `b` is the successor of `a`. -/
def is_succ: Rel ℕ ℕ
  | a, b ⇒ a + 1 = b

/-- The RTC of `is_succ` is equivalent to `≤` -/
lemma rtc_is_succ_iff_le (a b: ℕ): ReflTransGen is_succ a b ↔ a ≤ b :=
  /- proof omitted -/
```

In our final definition of ARS, we therefore use `Rel α α` instead of `Set (α × α)`:

Definition 3.4 (Our final definition of `ARS`).

```
structure ARS (α I: Type) where
  rel: I → Rel α α
```

As is the case for our set-theoretic definition, we do not have a separate definition for a non-indexed ARS; using Definition 3.4, we can still represent a non-indexed ARS by using the single-inhabitant `Unit` type as our index type.

Aside from the main definition, we define some notation for common closures over relations, as well as a few derived relations: the union of rewrite relations and the convertability relation of an ARS:

Definition 3.5 (Closure notation and derived relations).

```
postfix:max (priority := high) "∗" ⇒ ReflTransGen
postfix:max (priority := high) "≡" ⇒ EqvGen
postfix:max (priority := high) "⁼" ⇒ ReflGen
postfix:max (priority := high) "⁺" ⇒ TransGen

/-- Two elements `a`, `b` are in the union of the rewrite relations of
    an ARS if there exists some index `i` s.t. `a →ᵢ b`. -/
abbrev ARS.union_rel (A: ARS α I): Rel α α :=
  fun a b ↦ ∃i, A.rel i a b

/-- The convertability relation for an ARS `A` is the equivalence closure
    of the union of its rewrite relations. -/
abbrev ARS.conv (A: ARS α I): Rel α α :=
  A.union_rel≡
```

Using the notation, we can for instance take the reflexive-transitive closure of a relation `r` simply by writing `r∗`, just as we could write $\rightarrow^*$ for the reflexive-transitive closure of $\rightarrow$ in traditional mathematical notation.

3.2.1 Reduction sequences

Now that we know how to represent reduction relations, we can continue with Definition 3.1 and consider how to represent reduction sequences.

The natural representation of reduction sequences in Lean is not immediately obvious. On one hand, finite reduction sequences are very similar to finite lists, which are represented inductively in Lean, so a natural representation might use inductive types. For instance, [11] uses a definition similar to the following (translated from Isabelle):

Definition 3.6 (Inductive reduction sequence)**.** The inductive definition of a reduction sequence consists of the following introduction rules:

- There is an empty reduction sequence from any x to itself.
- If $x \longrightarrow y$ is a reduction step, and there is a reduction sequence from y to z, there is a reduction sequence from x to z.

```
inductive ReductionSeq: α → α → List (α × α) → Prop
  | refl {x} : ReductionSeq x x []
  | head {x y z ss} : r x y → ReductionSeq y z ss → ReductionSeq x z ((x, y)::ss)
```

A value of type `ReductionSeq r x y ss` represents a reduction sequence with respect to `r` from `x` to `y` with intermediate steps as given in `ss`. Such a value can be constructed by starting with the empty reduction sequence from `y` to `y`, `ReductionSeq.refl`, and prepending steps using `ReductionSeq.head`. For instance, given the individual steps $a \longrightarrow b$, $b \longrightarrow c$, $c \longrightarrow d$ we could construct the reduction sequence $a \longrightarrow b \longrightarrow c \longrightarrow d$ as follows:

Example 3.2 (A reduction sequence from $a \longrightarrow b \longrightarrow c \longrightarrow d$)**.**

```
open ReductionSeq -- so head, refl are in scope

example (s₁: r a b) (s₂: r b c) (s₃: r c d):
    ReductionSeq r a d [(a,b),(b,c),(c,d)] :=
  head s₁ (head s₂ (head s₃ refl))
```

An alternative definition appears when we consider the case of an infinite reduction sequence. We cannot use an inductive type to represent an infinite sequence, so instead, we turn to functions. An infinite sequence can be represented as a function from the natural numbers to the elements of the sequence, and if we add a requirement that these elements are linked by reduction steps, we have a definition of an infinite reduction sequence. In Lean, we can express this property as follows:

Definition 3.7 (Infinite reduction sequence)**.**

```
def inf_reduction_seq (f: ℕ → α) :=
  ∀n: ℕ, r (f n) (f (n + 1))
```

We can adapt this definition to include finite reduction sequences by adding a bound N.

Definition 3.8 (Generic (finite or infinite) reduction sequence).

```
def reduction_seq (N: ℕ∞) (f: ℕ → α) :=
  ∀n: ℕ, n < N → r (f n) (f (n + 1))
```

For this bound, we use the extended natural type (ℕ∞), which extends the natural numbers with a greatest element ⊤. This way, a function f that satisfies `reduction_seq ⊤ f` represents an infinite reduction sequence, while a function g that satisfies, say, `reduction_seq 12 g`, represents a finite (length-12) reduction sequence.

Our inductive definition has its advantages. For one, many proofs about finite reduction sequences proceed naturally using structural induction; for instance, the property that two reduction sequences can be concatenated to form a larger reduction sequence can be proved in only a few lines of trivial Lean:

```
lemma concat (h₁ : ReductionSeq r x y ss) (h₂: ReductionSeq r y z ss'):
    ReductionSeq r x z (ss ++ ss') := by
  induction h₁ with
  | refl              ⇒ exact h₂
  | head hstep _ ih ⇒ apply head hstep (ih h₂)
```

The same property using `reduction_seq` is more difficult to both state and prove:

```
def fun_aux (N: ℕ) (f g: ℕ → α): ℕ → α :=
  fun n ↦ if (n ≤ N) then f n else g (n - N)

def reduction_seq.concat (N₁ N₂: ℕ)
    (hseq: reduction_seq r N₁ f) (hseq': reduction_seq r N₂ g)
    (hend: f N₁ = g 0):
    reduction_seq r (N₁ + N₂) (fun_aux N₁ f g) := by
  intro n hn
  simp [fun_aux]
  norm_cast at *
  split_ifs
  · -- case within hseq
    apply hseq n (by norm_cast)
  · -- case straddling hseq and hseq'
    have: n = N₁ := by omega
    aesop
  · -- invalid straddling case (n > N₁, n + 1 ≤ N₁)
    omega
  · -- case within hseq'
```

```
    convert hseq' (n - N₁) (by norm_cast; omega) using 2
    omega
```

In many cases, however, we will have to deal with arbitrary reduction sequences, without knowing whether they are finite or infinite – for instance, the cofinality property, which plays a key part in our main result, guarantees the existence of cofinal reduction sequences, which can be finite or infinite. When defining and using such properties, it is convenient to have a unified definition of finite and infinite reduction sequences. Luckily, the fact that these definitions are equivalent in the finite case is easy to prove, so we can pick a representation at will and convert it if necessary.

Both definitions can be extended to represent indexed reduction sequences. In many cases, however, this is unnecessary, and using reduction sequences with respect to the union of rewrite relations is sufficient.

We will occasionally refer to the start, end, and elements of a `reduction_seq`; these are defined as follows:

Definition 3.9 (Start, end, and elements of a reduction sequence).

```
def reduction_seq.start (hseq: reduction_seq r N f) :=
  f 0

def reduction_seq.end (N: ℕ) (hseq: reduction_seq r N f) :=
  f N

def reduction_seq.elems (hseq: reduction_seq r N f): Set α :=
  f '' {x | x < N + 1} -- `f '' A` is the image of A under f
```

Note that the definition of `reduction_seq.end` requires `N` to be a natural number, i.e. the sequence to be finite.

3.2.2 Sub-ARSs

Occasionally, we might want to look at only part of an ARS. For instance, we might want to look at the ARS containing all elements that are reachable from some starting element `a` (the *reduction graph* of `a`), or at the ARS containing all elements that are convertible to an element `a` (the *component* of `a`). These partial ARSs are represented by the notion of a *sub-ARS*.

Definition 3.10 (sub-ARS, [8, p. 9], modified). Let $\mathcal{A} = (A, \rightsquigarrow_{i\in I})$ and $\mathcal{B} = (B, \rightarrow_{i\in I})$ be two ARSs. Then $\mathcal{A}$ is a *sub-ARS* of $\mathcal{B}$, denoted $\mathcal{A} \subseteq \mathcal{B}$, if the following conditions are satisfied:

(i) $A \subseteq B$;

(ii) For all $i \in I$, $\rightsquigarrow_i$ is the restriction of $\to_i$ to A, i.e. $\rightsquigarrow_i \; = \; \to_i \; \cap A^2$;

(iii) For all $i \in I$, A is closed under $\to_i$, i.e. $\forall a \in A, b \in B, (a \to_i b \Rightarrow b \in A)$.

```
/-- If `S: SubARS B`, `S` is a sub-ARS of B. -/
structure SubARS (B: ARS β I) where
  /-- This SubARS contains the elements in the subtype `{b // p b}`. -/
  p: β → Prop
  /-- The ARS of this SubARS. -/
  ars: ARS {b: β // p b} I
  /-- `SubARS.ars.rel i` is the _restriction_ of `B.rel i` to the subtype. -/
  restrict: ∀(i: I) (a b: {b // p b}), ars.rel i a b ↔ B.rel i a b
  /-- `{b // p b}` is _closed_ under `B.rel i` -/
  closed: ∀(i: I) (a b: β), p a ∧ B.rel i a b → p b
```

The notion of a sub-ARS in [8] is only defined for non-indexed ARSs; we extend the definition here to encompass indexed ARSs, as long as the base ARS and sub-ARS share the same index type. We do this by requiring the restriction and closure properties to hold for each individual rewrite relation. Our initial definition instead required the restriction and closure properties to hold for the union of rewrite relations, and allowed reduction-equivalent ARSs to have a sub-ARS relationship. Mixing this notion of reduction equivalence and sub-ARSs later turned out to be problematic, because it makes the sub-ARS definition too weak, so they have been disentangled in the final definition.

In Lean, we choose to represent a sub-ARS as a separate structure, `SubARS`, which contains an ARS (`ars`) with the additional properties `restrict` and `closed`. To translate the subset requirement $A \subseteq B$, we require the elements of this ARS to have the type `{b // p b}`, a subtype of β.

We translate the restriction property by requiring `ars.rel` and `B.rel` to be equivalent for all `i: I` and `a b: {b // p b}`. Essentially, the intersection operation $\cap A^2$ is moved into the type system, and we are left with $\rightsquigarrow_i \; = \; \to_i$, which by propositional and functional extensionality is equivalent to $a \rightsquigarrow_i b \Leftrightarrow a \to_i b$. Note that Lean does not complain when we pass `a: {b // p b}` to `B.rel`, which expects a value of type β; this is because Lean automatically inserts a *coercion* from a subtype to the base type.

The closure property also has a slightly different form; instead of $a \in A, b \in B$, we let `a b: β` and require `a` to satisfy the subtype predicate `p`.

This is only one possible definition of a sub-ARS. Especially salient is the choice of which fields to make part of the type, and which not to. My initial definition kept the subtype property `p` in the type, just like `Subtype` does. This means distinct sub-ARSs have different types, which can be inconvenient in cases where you want to construct, for instance, a set of sub-ARSs, since all elements in a set need to have the same type. In the end, I decided to keep the property out of the type for this reason.

Now, of course, the downside of using subtypes is that it is not immediately obvious that this definition is equivalent to Definition 3.10. This is one of the pitfalls of interactive theorem proving: if we incorrectly formalize a definition, we might think we have proved something while actually proving something subtly different. Hopefully, careful consideration will convince the reader that our translation is indeed sensible.

Lemma 3.3. *The restriction and closure properties of a sub-ARS are respected by the union of rewrite relations, as well as the reflexive-transitive closure.*

```
variable (S: SubARS A)

lemma SubARS.restrict_union:
  ∀a b, S.ars.union_rel a b ↔ A.union_rel a b := /- proof omitted -/

lemma SubARS.closed_union:
  ∀a b, S.p a ∧ A.union_rel a b → S.p b := /- proof omitted -/

lemma SubARS.star_restrict:
  ∀i a b, (S.ars.rel i)∗ a b ↔ (A.rel i)∗ a b := /- proof omitted -/

lemma SubARS.star_closed:
  ∀i a b, S.p a ∧ (A.rel i)∗ a b → S.p b := /- proof omitted -/
```

3.2.3 Reduction graphs and components

We immediately use our sub-ARS definition to define the notions of *reduction graph* and *component*.

Definition 3.11 (Reduction graph and component)**.** For all $A \subseteq B$ we can generate a sub-ARS $\mathcal{G}(A, \rightarrow_j)$ of $\mathcal{B}$, which contains exactly all elements reachable using $\rightarrow_j$ from an element in A. In particular, we are often interested in the sub-ARS generated by a single element $b \in B$; this is also called the *reduction graph* of b, sometimes abbreviated as $\mathcal{G}(b)$.

```
/-- The sub-ARS generated by a set of elements of β -/
def SubARS.generate (B: ARS β I) (s: Set β) : SubARS B where
  p := (fun b ↦ ∃a, a ∈ s ∧ B.union_rel∗ a b)
  ars := ⟨fun i a b ↦ B.rel i a b⟩
  restrict := /- proof omitted -/
  closed := /- proof omitted -/

/-- The reduction graph of `b` in `B` consists of all reducts of `b` -/
def ARS.reduction_graph (B: ARS β I) (b: β) : SubARS B :=
  SubARS.generate B {b}
```

A (connected) *component* of an element $a \in A$ is the sub-ARS containing all elements $b \equiv a$, with the reduction relation restricted to this convertability class. In Lean, we extend the `SubARS` structure to form a `Component` structure, which has the additional properties that you would expect for a component: a connected component is a nonempty set of elements, all of which are related by the convertibility relation, closed under the convertibility relation.

```
@[ext]
structure Component extends SubARS A where
  component_restrict: ∀{a b}, p a → p b → A.conv a b
  component_closed: ∀{a b}, p a → A.conv a b → p b
  component_nonempty: ∃a, p a

def ARS.component (a: α): Component A where
  p := (A.conv a ·)
  ars := ⟨fun i a b ↦ A.rel i a b⟩
  restrict := /- proof omitted -/
  closed := /- proof omitted -/
  component_restrict := /- proof omitted -/
  component_closed := /- proof omitted -/
  component_nonempty := /- proof omitted -/
```

Now that we have defined the basic structures in abstract rewriting, we can begin to define the key properties of rewrite relations. In the following sections, we present a number of definitions related to confluence and normalization, and we will end chapter 3 by defining a few miscellaneous properties.

3.3 Confluence

In this section, we present the definitions related to confluence that are used in the rest of this thesis. So we can compare and contrast the two, we have interspersed the traditional definitions with our Lean definitions.

Definition 3.12 (confluence, [8, p. 10]). Let $\mathcal{A} = (A, \{\rightarrow_i, \rightarrow_j\})$ be an ARS. In Lean, let α be a type of objects, and `r s: Rel α α` be two rewrite relations on α.

(i) If $\forall a, b, c \in A, (c \leftarrow_j a \rightarrow_i b \implies \exists d \in A, (c \twoheadrightarrow_i d \twoheadleftarrow_j b))$, we say $\rightarrow_i$ *commutes weakly* with $\rightarrow_j$.

```
def weakly_commutes :=
  ∀a b c, r a b ∧ s a c → ∃d, s* b d ∧ r* c d
```

(ii) If $\twoheadrightarrow_i$ and $\twoheadrightarrow_j$ commute weakly, we say $\rightarrow_i$ and $\rightarrow_j$ *commute*.

```
def commutes :=
  ∀a b c, r* a b ∧ s* a c → ∃d, s* b d ∧ r* c d
```

(iii) Let $a \in A$. If $\forall b, c \in A, (c \leftarrow_i a \rightarrow_i b \Rightarrow \exists d \in A, (c \twoheadrightarrow_i d \twoheadleftarrow_i b))$, we say a is *weakly confluent* with respect to $\rightarrow_i$. The reduction relation $\rightarrow_i$ is weakly confluent or *weakly Church-Rosser* (WCR) if every $a \in A$ is weakly confluent. Weak confluence is also called *local confluence*.

```
def weakly_confluent :=
  ∀a b c, r a b ∧ r a c → ∃d, r* b d ∧ r* c d
```

(iv) Let $a \in A$. If $\forall b, c \in A, (c \leftarrow_i a \rightarrow_i b \Rightarrow \exists d \in A, (c \rightarrow_i^= d \leftarrow_i^= b))$, we say a is *subcommutative* with respect to $\rightarrow_i$. The reduction relation $\rightarrow_i$ is subcommutative ($\mathrm{CR}^{\leq 1}$) if every $a \in A$ is subcommutative.

```
def subcommutative :=
  ∀a b c, r a b ∧ r a c → ∃d, r⁼ b d ∧ r⁼ c d
```

(v) Let $a \in A$. If $\forall b, c \in A, (c \leftarrow_i a \rightarrow_i b \Rightarrow \exists d \in A, (c \rightarrow_i d \leftarrow_i b))$, we say a has the *diamond property* (DP) with respect to $\rightarrow_i$. The reduction relation $\rightarrow_i$ has the diamond property if every $a \in A$ has the diamond property.

```
def diamond_property :=
  ∀a b c, r a b ∧ r a c → ∃d, r b d ∧ r c d
```

(vi) Let $a \in A$. If $\forall b, c \in A, (c \twoheadleftarrow_i a \twoheadrightarrow_i b \Rightarrow \exists d \in A, (c \twoheadrightarrow_i d \twoheadleftarrow_i b))$, we say a is *confluent* with respect to $\rightarrow_i$. The reduction relation $\rightarrow_i$ is confluent or *Church-Rosser* (CR) if every $a \in A$ is confluent.

```
def confluent :=
  ∀a b c, r* a b ∧ r* a c → ∃d, r* b d ∧ r* c d
```

Most of these definitions have both local (per-element) and global (for the set of all elements) versions. The given Lean definitions are for the global versions, but there are separate local versions where necessary, which have a prime symbol added to their name. For example, `confluent' r a` means the element `a` is confluent with respect to `r`.

Since these properties are defined on the individual rewrite relations, we do not use the ARS structure in Lean – we will see it appear again later, however. Additionally, note that we do not need to give a type ascription for `a`, `b`, `c`, `d` in Lean; since our relations are typed, Lean infers that these variables must have type `α`.

Other than these technicalities, the Lean definitions are essentially direct translations. The only exception is our definition of commutation, which is defined directly instead of being based on weak commutation. Note that our direct definition makes use of the fact that `r** = r*`, i.e. taking the reflexive-transitive closure is idempotent.

A few properties are globally equivalent to confluence, and are often used in confluence proofs. These are listed in Definition 3.13.

Definition 3.13 (Properties equivalent to confluence)**.** Let $\mathcal{A} = (A, \to)$ be an ARS. In Lean, let `r: Rel α α` be a rewrite relation on `α`.

(i) If $\forall a, b, c \in A, (c \leftarrow a \twoheadrightarrow b \Rightarrow \exists d \in A, (c \twoheadrightarrow d \twoheadleftarrow b))$, we say the relation $\to$ is *semi-confluent*.

```
def semi_confluent :=
  ∀a b c, r* a b ∧ r a c → ∃d, r* b d ∧ r* c d
```

(ii) If $\forall a, b \in A, a \equiv b \Rightarrow \exists c \in A, a \twoheadrightarrow c \twoheadleftarrow b$, we say the relation $\to$ is *conversion confluent*.

```
def conv_confluent :=
  ∀a b, (r≡) a b → ∃c, r* a c ∧ r* b c
```

Lemma 3.4. *Global confluence, semi-confluence, and conversion confluence are equivalent.*

Proof. See `semi_confluent_iff_confluent` and `conv_confluent_iff_confluent`.

Note that conversion confluence is often referred to as the Church-Rosser property. Since [8] uses that name interchangeably with confluence, we use the separate name conversion confluence.

For an example of how to use the Lean definitions, let us consider the `is_succ` definition from Example 3.1. We can prove that `is_succ` is confluent:

```
example: confluent is_succ := by
  rintro a b c ⟨hab, hac⟩
  show ∃d, is_succ* b d ∧ is_succ* c d
  · use (max b c)

    show is_succ* b (max b c) ∧ is_succ* c (max b c)

    suffices b ≤ (max b c) ∧ c ≤ (max b c) by
      rwa [rtc_is_succ_iff_le, rtc_is_succ_iff_le]

    show b ≤ (max b c) ∧ c ≤ (max b c)
    · omega
```

Of course, this is a bit of a contrived example, as there is no real divergence here, and we don't need the hypotheses `hab: is_succ* a b` and `hac: is_succ* a c`. Nevertheless, it shows the main elements of a confluence proof: to show that a relation `r` is confluent, we take an arbitrary `a, b, c` and proofs that `r* a b` and `r* a c`, and we must show that there is a `d` such that `r* b d` and `r* c d`.

3.4 Normalization

In this section, we present various definitions related to normalization. As with our confluence definitions, we intersperse the 'on paper' definitions with Lean definitions.

Definition 3.14 (normalization). Let $\mathcal{A} = (A, \to)$ be an ARS.

(i) $a \in A$ is a *normal form* if there exists no $b \in A$ such that $a \to b$.

```
def normal_form (a: α) :=
  ¬∃b, r a b
```

(ii) $a \in A$ is *weakly normalizing* (WN) if $a \twoheadrightarrow b$ for some normal form $b \in A$. The reduction relation $\to$ is weakly normalizing if every $a \in A$ is weakly normalizing.

```
def weakly_normalizing :=
  ∀a, ∃b, r∗ a b ∧ normal_form r b
```

(iii) $a \in A$ is *strongly normalizing* (SN) if there are no infinite reduction sequences starting from a. The reduction relation $\to$ is strongly normalizing if every $a \in A$ is strongly normalizing.

```
def strongly_normalizing :=
  ¬∃(f: ℕ → α), reduction_seq r ⊤ f
```

(iv) If $a \equiv b \Rightarrow a \twoheadrightarrow b$ for all $a \in A$ and any normal form $b \in A$, we say $\to$ has the *normal form property* (NF).

```
def normal_form_property :=
  ∀a b, normal_form r b → (r≡) a b → r∗ a b
```

(v) If $a \equiv b \Rightarrow a = b$ for all normal forms $a, b \in A$, we say $\to$ has the *unique normal form property* (UN).

```
def unique_normal_form_property :=
  ∀a b, normal_form r a → normal_form r b → (r≡) a b → a = b
```

(vi) If, for every $c \in A$ that reduces to two normal forms $a, b \in A$, we have $a = b$, we say $\to$ has the *unique normal form property with respect to reduction* ($\text{UN}^{\to}$).

```
def unique_normal_form_property_r :=
  ∀c a b, normal_form r a → normal_form r b → r∗ c a → r∗ c b → a = b
```

The most notable difference in our Lean definitions is in our definition of strong normalization; instead of defining a relation as strongly normalizing if all elements are strongly normalizing, we simply define a relation to be strongly normalizing if it admits no infinite rewrite sequences. We again have separate per-element definitions of weak and strong normalization (`weakly_normalizing'` and `strongly_normalizing'`).

Strong normalization is related to another property of binary relations: well-foundedness. There are a number of formulations of well-foundedness, which are equivalent if one assumes the axiom of choice. The Lean definition is as follows:

Definition 3.15 (Well-foundedness).

```
/-- The accessibility predicate. If `Acc r x`, `x` is accessible through `r`. -/
inductive Acc {α} (r: α → α → Prop): α → Prop where
  /-- A value is accessible if all of its descendants are also accessible. -/
  | intro (x: α) (h: (y: α) → r y x → Acc r y): Acc r x

/-- A relation `r: α → α → Prop` is well-founded if all elements of `α`
    are accessible within `r`. -/
inductive WellFounded (r: α → α → Prop): Prop where
  | intro (h: ∀x, Acc r x): WellFounded r
```

The main application of well-foundedness is *well-founded induction*: if we have a well-founded relation `r: α → α → Prop` and a predicate `p: α → Prop`, in order to prove `∀x: α, p x`, it suffices to prove `∀x, (∀y, r y x → p y) → p x`. If we let `α := ℕ` and `r := (fun a b ↦ a < b)`, we get strong induction on natural numbers; well-founded induction is a generalization of strong induction to other well-founded relations. In the case of Lean, the principle of well-founded induction follows from structural induction on the accessibility predicate.

The above definition is equivalent to these two alternative definitions:

Definition 3.16 (Well-foundedness, alternative).

(i) If a relation R on α is well-founded, there are no infinitely descending sequences $\cdots R\, y_2\, R\, y_1\, R\, y_0$ (see the proof of Theorem 3.5, below).

(ii) If a relation R on α is well-founded, it has a minimum on every set X of elements of α, i.e. there is some element $x \in X$ s.t. $\forall y, \neg yRx$. (`WellFounded.wellFounded_iff_has_min`)

While the similarity is non-obvious from Definition 3.15, Definition 3.16(i) is almost equivalent to strong normalization: if a relation is strongly normalizing, there are no infinitely *ascending* sequences $y_0\, R\, y_1\, R\, y_2\, R\, \cdots$; this leads us to the following correspondence between strong normalization and well-foundedness.

Theorem 3.5. *A relation R is strongly normalizing if and only if its inverse is well-founded.*

Note that our proof below only assumes Definition 3.15; if we assume equivalence to Definition 3.16(i), the proof is immediate.

Proof.

Case $\Rightarrow$ We take the contrapositive. Assume R^{-1} is not well-founded. Then there is an element which is not accessible. For any inaccessible element x, there must be an element y with $yR^{-1}x$ $(= xRy)$ which is also inaccessible. This allows us to build an infinite ascending R-chain of inaccessible elements, contradicting strong normalization.

Case $\Leftarrow$ Assume R^{-1} is well-founded. We wish to show that R is strongly normalizing, i.e. there are no infinite ascending R-chains. Assume there exists some element $a : \alpha$ – we can freely do so, because if α is uninhabited, R is certainly strongly normalizing. We must show that there can be no infinite sequence $f : \mathbb{N} \to \alpha$ starting with a which is an R-chain, i.e.

$$\forall f, f(0) = a \Rightarrow \neg\forall n, f(n)\, R\, f(n+1)$$

We proceed by well-founded induction on a. We must then prove

$$\begin{aligned}\forall a, (\forall y, yR^{-1}a \Rightarrow \forall f, f(0) = y \Rightarrow \neg\forall n, f(n)\, R\, f(n+1))\\ \Rightarrow (\forall f, f(0) = a \Rightarrow \neg\forall n, f(n)\, R\, f(n+1))\end{aligned}$$

Fix a, the induction hypothesis, and f, and assume $f(0) = a$. We take the contrapositive with respect to the induction hypothesis. We then have to prove

$$(\forall n, f(n)\, R\, f(n+1)) \Rightarrow \exists y, yR^{-1}a \wedge \exists g, g(0) = y \wedge \forall n, g(n)\, R\, g(n+1)$$

Assume $\forall n, f(n)\, R\, f(n+1)$. Let $y := f(1)$. We have $yR^{-1}a = f(0)\, R\, f(1)$ by assumption. Let $g(n) := f(n+1)$. We have $g(0) = f(1) = y$ by definition, and $\forall n, g(n)\, R\, g(n+1)$ by assumption. □

Linking strong normalization to Lean's notion of well-foundedness also gives us an easy proof that strong normalization implies weak normalization:

Lemma 3.6. *Any strongly normalizing relation R is weakly normalizing.*

Proof. Let a be an element in A. We wish to show that a is weakly normalizing, i.e. a has a reduct that is a normal form. Since R is strongly normalizing, its inverse is well-founded, and thus has a minimum on any non-empty subset of A. Let $X := \{\ b \mid aR^*b\ \}$ be the set of reducts of a. X is nonempty, since a is a 0-step reduct of a. Then R^{-1} has a minimum on X, that is, $\exists x \in X, \forall y, \neg yR^{-1}x$. Since $x \in X$, x is a reduct of a, and since $\forall y, \neg xRy$, x is a normal form. □

3.5 Miscellaneous properties

Although confluence and normalization are the most important basic ARS properties, there are a few miscellaneous properties which are collected here.

Definition 3.17 (miscellaneous).

(i) A relation is *complete* if it is confluent and strongly normalizing.

```
def complete
  := confluent r ∧ strongly_normalizing r
```

(ii) A relation is *semi-complete* if it has the unique normal form property and is weakly normalizing.

```
def semi_complete :=
  unique_normal_form_property r ∧ weakly_normalizing r
```

(iii) A relation is *inductive* if, for every reduction sequence, there exists an element a that is a reduct of every element in the reduction sequence.

```
def rel_inductive :=
  ∀{N f} (hseq: reduction_seq r N f), ∃a, ∀b ∈ hseq.elems, r⋆ b a
```

(iv) A relation is *increasing* if there exists a function $f : \alpha \to \mathbb{N}$ which increases with every reduction step.

```
def increasing :=
  ∃(f: α → ℕ), ∀{a b}, r a b → f a < f b
```

3.6 Interrelations between ARS properties

As shown in Fig. 2, many of the basic properties on ARSs are interrelated. Most of these implications are intuitive translations of the informal proofs, and will not be discussed further. Instead, in the sequel, we will focus on a few key properties and theorems (Newman's Lemma, cofinality, Decreasing Diagrams) and discuss these in detail.

4 Newman's Lemma

Newman's Lemma is perhaps the first important, non-trivial result in abstract rewriting, giving a sufficient condition for weak confluence to imply confluence, namely strong normalization.

Theorem 4.1 (Newman's Lemma). *Every strongly normalizing, weakly confluent reduction relation is confluent.*

There are various proofs of Newman's Lemma in the literature, ranging in complexity. We will discuss the three proofs contained in [8], as they provide an interesting case study on how different proofs of the same theorem may be more or less amenable to formalization in a proof assistant.

4.1 Barendregt's proof

The first proof, due to Barendregt [1, Proposition 3.1.25], is by far the easiest to translate to Lean. It shows confluence via uniqueness of normal forms, by constructing an infinite reduction sequence of elements having multiple distinct normal forms (so-called *ambiguous* elements), which contradicts our assumption of strong normalization.

Our formalization bears many similarities to the pen-and-paper proof in [8, p. 15], but makes explicit some steps that are deemed obvious in that proof.

Lemma 4.2. *Weak normalization and uniqueness of normal forms imply confluence.*

```
lemma confluent_of_wn_unr (hwn: weakly_normalizing r) (hun: unique_nf_prop_r r):
    confluent r := ...
```

Proof. Fix a, b, c and assume $a \to^* b$ and $a \to^* c$. By weak normalization, b and c have normal forms nf_b and nf_c, such that $b \to^* nf_b$ and $c \to^* nf_c$. Since b and c are reducts of a, these are also normal forms of a. By the unique normal form property w.r.t. reduction, we must have $nf_b = nf_c$. □

Lemma 4.3. *If an element a has two distinct normal forms, the reduction sequence from a to each normal form is at least one step long.*

```
lemma trans_step_of_two_normal_forms {r: Rel α α} {a d₁ d₂: α}
    (hd₁: normal_form r d₁) (hd₂: normal_form r d₂)
    (had₁: r∗ a d₁) (had₂: r∗ a d₂) (hne: d₁ ≠ d₂): r⁺ a d₁ ∧ r⁺ a d₂ := ...
```

Proof. We proceed by contradiction; assume at least one of d_1 and d_2 has an empty reduction sequence from a – that is, at least one of d_1, d_2 is equal to a.

If both elements are equal to a, they are not distinct. Alternatively, without loss of generality, let d_1 be equal to a, d_2 be distinct from a. Then d_1 cannot be a normal form; it has a reduct (d_2). □

Definition 4.1. An *ambiguous* element is an element with at least two distinct normal forms.

```
def ambiguous (a: α) :=
  ∃(b c: α), r* a b ∧ r* a c ∧ normal_form r b ∧ normal_form r c ∧ b ≠ c
```

Lemma 4.4. *If r is weakly normalizing and weakly confluent, any element that is ambiguous in r has a one-step reduct which is also ambiguous.*

```
lemma ambiguous_reduct_of_ambiguous
    (hwn: weakly_normalizing r) (hwc: weakly_confluent r):
    ∀a, ambiguous r a → ∃b, r a b ∧ ambiguous r b := ...
```

Proof. Assume a is ambiguous. Then it has at least two distinct normal forms d_1 and d_2. By Lemma 4.3, we must have $a \to b \twoheadrightarrow d_1$ and $a \to c \twoheadrightarrow d_2$ for some b, c.

By weak confluence, b and c have a common reduct, d, which by weak normalization has a normal form, nf_d. nf_d must be distinct from at least one of d_1, d_2; without loss of generality, say $nf_d \neq d_1$. Then b, having two distinct normal forms, is our desired ambiguous one-step reduct. □

Lemma 4.5. *Any weakly confluent relation that does not have the unique normal form property w.r.t. reduction is not strongly normalizing.*

```
lemma not_sn_of_wc_not_un (hwc: weakly_confluent r) (hnu: ¬unique_nf_prop_r r):
    ¬strongly_normalizing r := by
```

Proof. Assume r is weakly confluent, but does not have the unique normal form property w.r.t. reduction. We may also freely assume r is weakly normalizing; if not, it certainly isn't strongly normalizing.

Since r does not have the unique normal form property, it must have an element which has two distinct normal forms, i.e. an ambiguous element.

Using Lemma 4.4, we can build an infinite chain of ambiguous reducts, contradicting strong normalization. □

Lemma 4.6 (Newman). *Any strongly normalizing, weakly confluent relation r is confluent.*

```
lemma newman (hsn: strongly_normalizing r) (hwc: weakly_confluent r):
    confluent r := ...
```

Proof. Assume r is strongly normalizing and weakly confluent. By Lemma 3.6, r is weakly normalizing. Then, by Lemma 4.2, it suffices for confluence to prove that r has the unique normal form property w.r.t. reduction, which is obvious from Lemma 4.5. □

Aside from the definition of an ambiguous element, which is easy to translate into Lean, this uses only definitions we already have. We need a small auxiliary lemma (uniqueness of normal forms and weak normalization imply confluence), but other than that the Lean proof is very straightforward.

4.2 Proof by well-founded induction

The second proof for Newman's Lemma is by well-founded induction. We only need one small auxiliary lemma, which already exists in *mathlib* as `WellFounded.transGen`:

Lemma 4.7. *If r is well-founded, so is its transitive closure r^+.*

```
lemma trans_wf_of_wf (hwf: WellFounded r): WellFounded r⁺ :=
  hwf.transGen
```

Lemma 4.8 (Newman). *Any strongly normalizing, weakly confluent relation $\longrightarrow$ is confluent.*

```
lemma newman₂ (hsn: strongly_normalizing r) (hwc: weakly_confluent r):
  confluent r := ...
```

Proof ([8, p. 15–16]). Since $\longrightarrow$ is strongly normalizing, $\longleftarrow$ is well-founded by Theorem 3.5. By Lemma 4.7, so is its transitive closure $\longleftarrow^+$.

Fix a. We wish to prove that a is confluent, that is $\forall bc, c \twoheadleftarrow a \twoheadrightarrow b \Longrightarrow \exists d, b \twoheadrightarrow d \twoheadleftarrow c$.

We proceed by well-founded induction on a, with respect to the transitive closure of the inverse of our relation $\longrightarrow$.

Our induction hypothesis is that all reducts of a are confluent, that is, $\forall a', a' \longleftarrow^+ a \Longrightarrow (\forall bc, c \twoheadleftarrow a' \twoheadrightarrow b \Longrightarrow \exists d, b \twoheadrightarrow d \twoheadleftarrow c)$.

Fix b and c, and assume $a \twoheadrightarrow b$ and $a \twoheadrightarrow c$. Without loss of generality, assume that both b and c are distinct from a; if not, one of them is our desired common reduct d. Then we have $c \twoheadleftarrow c' \longleftarrow a \longrightarrow b' \twoheadrightarrow b$ for some b', c'.

By weak confluence, b' and c' have a common reduct, call it d'. That is, $b' \twoheadrightarrow d' \twoheadleftarrow c'$. Since b' and c' are reducts of a, they are confluent by our induction hypothesis. Then, since d' and b are both reducts of b', they have a common reduct, call it e. Since c and e (via d') are both reducts of c', they have a common reduct, call it d. This d is a common reduct of b and c, just as we desire. ☐

$$\begin{tikzcd}
a \arrow[r] \arrow[d] \arrow[dr, phantom, "\text{WCR}"] & b' \arrow[r, twoheadrightarrow] \arrow[d, twoheadrightarrow] \arrow[dr, phantom, "\text{CR}(b')"] & b \arrow[d, twoheadrightarrow] \\
c' \arrow[r, twoheadrightarrow] \arrow[d, twoheadrightarrow] \arrow[drr, phantom, "\text{CR}(c')"] & d' \arrow[r, twoheadrightarrow] & e \arrow[d, twoheadrightarrow] \\
c \arrow[rr, twoheadrightarrow] & & d
\end{tikzcd}$$

Figure 3: An illustration of the proof of Newman's Lemma by well-founded induction.

This proof is perhaps the most elegant in Lean, as it uses no auxiliary notions other than well-founded induction, which is already well-supported in Lean.

4.3 Proof by terminating peak-elimination

Since the third proof is considerably more complex, we provide a complete proof sketch here:

Proof sketch ([8, p. 16–17], modified). Assume $\rightarrow$ is strongly normalizing and weakly confluent. We wish to prove that $\rightarrow$ is confluent. By Lemma 3.4 it suffices to prove that $\rightarrow$ is conversion confluent, i.e., for all $a, b \in A$ that are equivalent, there exists a common reduct c.

Let $a \equiv b$. Then there exist $a_0, a_1, \dots, a_n \in A$ such that $a = a_0 \leftrightarrow \cdots \leftrightarrow a_n = b$. We view $a_0 \leftrightarrow \cdots \leftrightarrow a_n$ as a landscape, which may contain *peaks* $a_{i-1} \leftarrow a_i \rightarrow a_{i+1}$. If a landscape contains no peaks, then it must meet at some point which is reached by only taking forward steps from a, and only backward steps from b. This meeting point is then our desired common reduct c.

Assume our landscape *does* contain peaks. By weak confluence, we can eliminate a peak $a_{i-1} \leftarrow a_i \rightarrow a_{i+1}$, producing a valley $a_{i-1} \rightarrow c_1 \twoheadrightarrow d \twoheadleftarrow c_1' \leftarrow a_{i+1}$ (see Fig. 5). Note that this may create new peaks to the left and right of a_{i-1} and a_{i+1}, respectively, so it does not immediately help us.

The key observation is that we can show that, since $\rightarrow$ is strongly normalizing, repeating this peak-elimination procedure must terminate. This means we must end up with a landscape which contains no peaks, and therefore contains a common reduct c as described earlier.

This argument uses *multisets*: sets that may contain an element multiple times. To a landscape $a_0 \leftrightarrow \cdots \leftrightarrow a_n$ we associate the multiset $[a_0, \dots, a_n]$. As $\rightarrow$ is strongly normalizing, $\leftarrow$ is well-founded (Theorem 3.5), as is $\leftarrow^+$ (Lemma 4.7). We can extend $\leftarrow^+$ to a well-founded order on multisets of A, $\leftarrow_\#^+$. The idea is that, for two multisets M, N, we have $M \leftarrow_\#^+ N$ if M is derived from N by replacing one or more elements by smaller elements (w.r.t. $\leftarrow^+$). Since $\rightarrow$ is strongly normalizing, taking multisets that are smaller w.r.t. $\leftarrow_\#^+$ moves the elements in the multiset towards their normal forms, and since all reduction sequences are finite, $\leftarrow_\#^+$ must be well-founded.

We argue that performing peak elimination on a landscape M produces a new landscape M' with $M' \leftarrow_\#^+ M$. If $M := [a_0, \dots, a_{i-1}, a_i, a_{i+1}, \dots, a_n]$, then performing peak elimination on a_i yields the multiset $M' := [a_0, \dots, a_{i-1}, c_1, \dots, d, \dots, c_1', \dots, a_{i+1}, \dots, a_n]$. Note that the replacement elements, $[c_1, \dots, d, \dots, c_1']$, are all reducts of a_i by WCR. Therefore, $M' \leftarrow_\#^+ M$, and by well-foundedness of the multiset relation, our repeating peak-elimination procedure must terminate. $\square$

$$
\begin{gathered}
c = a = a_0 \twoheadleftarrow a_n = b \\
a = a_0 \twoheadrightarrow c \twoheadleftarrow a_n = b \\
a = a_0 \twoheadrightarrow a_n = b = c
\end{gathered}
$$

Figure 4: Landscapes without peaks, and their common reducts c.

$$\begin{tikzcd}
 & & & a_i \arrow[dll] \arrow[drr] & & & \\
a_0 \arrow[r, <<->>] & a_{i-1} \arrow[dr, dotted] & & & & a_{i+1} \arrow[dl, dotted] \arrow[r, <<->>] & a_n \\
 & & c_1 \arrow[dr, dotted] & & c_1' \arrow[dl, dotted] & & \\
 & & & d & & &
\end{tikzcd}$$

Figure 5: A single step in our peak-elimination procedure.

This proof is by far the most complex to translate to Lean. In part, this is because the proof requires a few idiosyncratic notions that are not yet formalized, such as landscapes and peaks. Multisets are available in Lean (`Multiset`), but the multiset extension of a relation and a proof of its well-foundedness are not. The proofs of the prerequisites already take up a few hundred lines of Lean code.

Aside from the cost of formalizing its prerequisites, the proof itself is simply not well-suited to Lean. Because of the geometric intuition behind the proof, it is well-suited to a pen-and-paper environment, where illustrations can help the reader along, and it is easy to see how sequences are split up into disjoint parts. In Lean, we have no such luck, and many of the steps that involve manipulating reduction sequences and multisets are very tedious. In the end, formalizing this proof took upwards of a week, whereas the other proofs could be formalized in a few hours at most.

We will detail the Lean proof below. We start by formalizing the multiset order extension and proving that it is well-founded if the underlying relation is well-founded. Then, we define various helper lemmas for finite symmetric reduction sequences, showing that such a sequence contains a common reduct of the endpoints if it contains no peaks. Armed with these prerequisites, we will show that a single step in our peak-elimination process decreases the multiset of elements in the sequence with respect to the multiset order. Lastly, we tie everything together by showing that, if our underlying relation is well-founded, this process must terminate.

4.3.1 Multisets

As mentioned, `Multiset`s are available in Lean, where they are modeled as a quotient type of lists by permutation. On these multisets, we can define the following relation, which extends a relation on the elements of the multiset.

Definition 4.2 (Dershowitz-Manna ordering, [8, p. 821]). Let $\leftarrow\ \subseteq A \times A$ be a relation on some set A. The multiset extension of $\leftarrow$, denoted $\leftarrow_{\#}$, is the smallest transitive relation satisfying

$$\text{if } \forall x \in M', x \leftarrow s, \text{ then } M + M' \leftarrow_{\#} M + \{s\} \tag{1}$$

In essence, a multiset decreases according to the relation when we replace an element with a subset of its reducts. This notion was first introduced by Dershowitz and Manna in [2], where it was shown to be a well-founded order on multisets over A if the underlying relation is a well-founded order over A. It is therefore often referred to as the *Dershowitz-Manna ordering*.

In Lean, we first define $\leftarrow_{\#}^{1}$, the smallest relation that satisfies Eq. (1), as an inductive type, and then define the Dershowitz-Manna ordering as the transitive closure of that relation.

```
inductive MultisetExt1 : Multiset α → Multiset α → Prop where
| rel (M M': Multiset α) (s: α) (h: ∀m ∈ M', r m s):
  MultisetExt1 (M + M') (s ::ₘ M)

abbrev MultisetExt := (MultisetExt1 r)⁺
```

The literature contains various proofs that $\leftarrow_{\#}$ is a well-founded order if the underlying relation $\leftarrow$ is well-founded, but these are generally based on notions that are intuitive on paper, but non-trivial to formalize. For instance, the proof in [8, p. 822–823] proceeds by assuming $\leftarrow_{\#}$ is not well-founded, and using an infinite $\leftarrow_{\#}$-descending sequence to construct an infinite yet finitely branching tree of elements, with edges (more or less) representing steps along the underlying relation $\leftarrow$. By Kőnig's Lemma, then, this tree must contain an infinite path, which corresponds to an infinite $\leftarrow$-decreasing sequence, which conflicts with $\leftarrow$ being well-founded.

Isabelle/HOL already contains a formalization of the Dershowitz-Manna ordering, along with a well-foundedness proof. This proof, which is very distinct from the traditional proofs in the literature, makes use of the specific definition of well-foundedness in Definition 3.15. It does not require any auxiliary notions, and is well-suited to formalization, also in Lean. The source of this proof was initially a mystery, but after doing some digital archeology, could be traced to a mailing list message by Tobias Nipkow [5], where he notes that it is due to Wilfried Buchholz, gives a PostScript version of the proof (available as a PDF here), and notes that it is especially well-suited to formalizing in a theorem prover. We have used this proof in our formalization.

Lemma 4.9. *If $\leftarrow$ is well-founded, all multisets are accessible under $\leftarrow^1_\#$.*

```
lemma all_accessible (hwf: WellFounded r) (M: Multiset α):
    Acc (MultisetExt1 r) M := ...
```

Proof. See [6]. □

Draft note

I would like to describe the proof in more detail, but I must say I don't really understand it. That is, if I perform the steps as described, it all checks out, but I struggle to get an intuition behind how it works. I could, of course, just reproduce the steps in Nipkow's PostScript proof, but it would be nice if there was an intuitive explanation to go along with it.

Lemma 4.10. *If $\leftarrow$ is well-founded, then $\leftarrow_\#$ is well-founded.*

```
lemma MultisetExt.wf (WellFounded r):
    WellFounded (MultisetExt r) := ...
```

Proof. By Lemma 4.7, it suffices to show that $\leftarrow^1_\#$ is well-founded, which is true by Lemma 4.9 and Definition 3.15. □

4.3.2 Landscapes

As noted in the proof sketch, if two elements are equivalent, there must exist a symmetric reduction sequence $a = a_0 \leftrightarrow a_1 \leftrightarrow \dots \leftrightarrow a_n = b$ between them, which we call a landscape. There are multiple ways of representing such a sequence; we used to have a separate type `SymmSeq` for them, but at present simply encode them as reduction sequences over the symmetric closure of $\longrightarrow$, `ReductionSeq (SymmGen r)`. Curiously, although *mathlib* contains the reflexive, transitive, reflexive-transitive, and equivalence closures, it does not contain the symmetric closure, so `SymmGen` is defined by us.

Lemma 4.11. *If $a \equiv b$, there exists a landscape $a = a_0 \leftrightarrow \cdots \leftrightarrow a_n = b$, and vice versa.*

```
lemma exists_iff_rel_conv:
    (r≡) x y ↔ ∃ss, ReductionSeq (SymmGen r) x y ss := ...
```

Proof. By structural induction on (in the forward case) `EqvGen` and (in the backward case) `ReductionSeq`. □

Definition 4.3. A landscape $a = a_0 \leftrightarrow \cdots \leftrightarrow a_n = b$ has a peak if it contains two steps $a_{i-1} \leftarrow a_i \rightarrow a_{i+1}$.

```
def has_peak (hseq: ReductionSeq (SymmGen r) x y ss) :=
  ∃(n: ℕ) (h: n < ss.length - 1), r ss[n].snd ss[n].fst ∧ r ss[n + 1].fst ss[n + 1].snd
```

Lemma 4.12. *A landscape $a = a_0 \leftrightarrow \cdots \leftrightarrow a_n = b$ that contains no peaks has one of three forms:*

1. *Only forward steps, i.e. $a \twoheadrightarrow b$,*
2. *Only backward steps, i.e. $b \twoheadrightarrow a$,*
3. *A set of forward steps, followed by a set of backward steps, i.e. $a \twoheadrightarrow d \twoheadleftarrow b$.*

```
lemma not_peak_cases (hseq: ReductionSeq (SymmGen r) x y ss) (hnp: ¬hseq.has_peak):
    ReductionSeq r x y ss ∨ ReductionSeq r y x (steps_reversed ss) ∨ ∃ss₁ ss₂, (
      ss = ss₁ ++ ss₂ ∧ ss₁ ≠ [] ∧ ss₂ ≠ [] ∧
      ∃z, (ReductionSeq r x z ss₁ ∧ ReductionSeq r y z (steps_reversed ss₂))
    ) := ...
```

Proof. By structural induction on the landscape. □

We use the helper function `steps_reversed` here to reverse the list of steps as well as each individual step, so a list of steps `[(a, b), (b, c)]` becomes `[(c, b), (b, a)]`, as one would expect when reversing a reduction sequence.

Corollary 4.13. *If there is a landscape $a = a_0 \leftrightarrow \cdots \leftrightarrow a_n = b$ that contains no peaks, a and b have a common reduct.*

```
lemma reduct_of_not_peak (hseq: ReductionSeq (SymmGen r) x y ss) (hnp: ¬hseq.has_peak):
    ∃d, r∗ x d ∧ r∗ y d := ...
```

Proof. Immediate from Lemma 4.12. □

Aside from these core lemmas, the proof requires a lot of auxiliary lemmas that help with manipulating sequences in a way that is obvious to humans. For instance, a landscape between a and b becomes a landscape between b and a by turning each forward step $x \rightarrow y$ into a backward step $y \leftarrow x$; we can take or drop any number of steps from a landscape to get another landscape; reversing the steps in a landscape twice yields the original steps; et cetera. We omit them here for brevity.

4.3.3 Peak elimination

Lemma 4.14. *If a landscape $a = a_0 \leftrightarrow \cdots \leftrightarrow a_n = b$ contains a peak, and $\rightarrow$ is weakly confluent, there must be another landscape $a = a'_0 \leftrightarrow \cdots \leftrightarrow a'_m = b$, for which the multiset of elements $[a'_0, \ldots, a'_m]$ is smaller (w.r.t. $\leftarrow_{\#}$) than the multiset of elements $[a_0, \ldots, a_n]$.*

```
lemma newman_step' (hwc: weakly_confluent r) (hseq: ReductionSeq (SymmGen r) x y ss)
    (hp: hseq.has_peak): ∃(ss': _) (hseq': ReductionSeq (SymmGen r) x y ss'),
      MultisetExt (r.inv)⁺ (Multiset.ofList hseq'.elems) (Multiset.ofList hseq.elems)
    := ...
```

This lemma is the meat of the argument. On paper, the argument is simple; in our proof sketch, it takes up a single paragraph. That said, the description in the sketch is subtly incorrect (can you spot the mistake?), and benefits from the reader's intuition about reduction sequences. Lean has no such intuition, and thus, this proof is quite long, requiring a lot of intermediate steps where we prove we can split up the landscape or multiset of elements in a certain way.

Proof. Let $x = x_0 \leftrightarrow \cdots \leftrightarrow x_n = y$ be a landscape, which has a peak $x_{i-1} \leftarrow x_i \rightarrow x_{i+1}$.

We consider the case where $x_{i-1} = x_{i+1}$ separately from the case where $x_{i-1} \neq x_{i+1}$.

If $x_{i-1} = x_{i+1}$, then the steps $x_{i-1} \leftarrow x_i \rightarrow x_{i+1}$ are superfluous, and can simply be eliminated, yielding a smaller multiset because we remove two elements x_i and x_{i+1}.

If not, we split up our landscape into two landscapes $x_0 \leftrightarrow \cdots \leftrightarrow x_{i-1}$ and $x_{i+1} \leftrightarrow \cdots \leftrightarrow x_n$. By weak confluence of x_i, there must be two sequences $x_{i-1} = c_1 \rightarrow \cdots \rightarrow d$ and $x_{i+1} = c'_1 \rightarrow \cdots \rightarrow d$. We can then reconstitute a landscape from x_0 to x_n by concatenation: $x_0 \leftrightarrow \cdots \leftrightarrow x_{i-1} \rightarrow \cdots \rightarrow d \leftarrow \cdots \leftarrow x_{i+1} \leftrightarrow \cdots \leftrightarrow x_n$.

The elements in this landscape can be derived from the original elements by removing x_i and replacing it with the reducts $c_1, \ldots, d, \ldots, c'_1$. Hence, the multiset of elements is smaller according to $\leftarrow_{\#}$. □

Armed with Lemma 4.14, we can prove Newman's Lemma as follows:

Proof. Assume $\rightarrow$ is weakly confluent and strongly normalizing. We wish to show that $\rightarrow$ is confluent. By Lemma 3.4, it suffices to show that $\rightarrow$ is conversion confluent.

Let $a \equiv b$. We must show that a and b have a common reduct, c. By Corollary 4.13, it suffices to show that there is a landscape between a and b that has no peaks.

By Lemma 4.11, there exists a landscape between a and b. That means the set M_L of multisets corresponding to the elements of a landscape between a and b is nonempty.

By Theorem 3.5 and Lemma 4.10, since $\rightarrow$ is strongly normalizing, $\leftarrow$ is well-founded, as is $\leftarrow_{\#}$. That means there must be a minimal multiset M corresponding to a landscape between a and b. This landscape satisfies our goal; it does not contain a peak, for if it did, by Lemma 4.14, there would exist an even smaller multiset in M, contradicting the assertion that M is minimal. □

5 Cofinality

One of the miscellaneous ARS properties omitted from Section 3.5 is the cofinality property; this is because it plays a key part in our main result, and deserves to be discussed in detail here.

Definition 5.1 (Cofinality). Let $\mathcal{A} = (A, \to)$ be an ARS. Let $B \subseteq A$.

(i) B is *cofinal* in $\mathcal{A}$ if every $a \in A$ reduces to an element in B: $\forall a \in A, \exists b \in B, a \twoheadrightarrow b$.

```
def cofinal (s: Set α) :=
  ∀a, ∃b ∈ s, r∗ a b
```

(ii) A reduction sequence $a_0 \to a_1 \to \cdots$ is cofinal in $\mathcal{A}$ if the set of elements in the sequence is cofinal in $\mathcal{A}$.

```
def cofinal_reduction {r: Rel α α} {N: ℕ∞} {f: ℕ → α} (hseq: reduction_seq r N f) :=
  cofinal r hseq.elems
```

(iii) $\mathcal{A}$ has the *cofinality property* (CP) if for every $a \in A$, there exists a corresponding reduction sequence $a = a_0 \to a_1 \to \cdots$ which is cofinal in $\mathcal{G}(a)$, the reduction graph of a.

```
def cofinality_property :=
  ∀a, ∃N f, ∃(hseq: reduction_seq (A.reduction_graph a).ars.union_rel N f),
    cofinal_reduction hseq ∧ hseq.start = a
```

(iv) $\mathcal{A}$ has the *componentwise cofinality property* ($\text{CP}^{\equiv}$) if for every $a \in A$, there exists a corresponding reduction sequence $a = a_0 \to a_1 \to \cdots$ which is cofinal in $\mathcal{C}(a)$, the *component* of a.

```
def cofinality_property_conv :=
  ∀a, ∃N f, ∃(hseq: reduction_seq (A.component a).ars.union_rel N f),
    cofinal_reduction hseq ∧ hseq.start = a
```

Note that the cofinality property is one of the few properties we define for an ARS instead of a 'raw' reduction relation – this is necessary because it uses the notion of a sub-ARS that we defined in Section 3.2.2.

5.1 Cofinality and confluence

Cofinality seems like a strange property, but it turns out to have an interesting connection to confluence. It follows almost directly from the definition that an ARS which has the cofinality property must be confluent:

Lemma 5.1. *An ARS $\mathcal{A} = (A, \to)$ which has the cofinality property must be confluent.*

```
lemma cr_of_cp {A: ARS α I}:
    cofinality_property A → confluent A.union_rel := ...
```

Proof. Let $a, b, c \in A$, and $c \twoheadleftarrow a \twoheadrightarrow b$. By the cofinality property, there is some reduction sequence $s_0 \to s_1 \to \ldots$ which is cofinal in the reduction graph of a.
Since b and c are in the reduction graph of a, they reduce to elements s_b, s_c in the cofinal reduction sequence. Without loss of generality, we take $s_b \twoheadrightarrow s_c$. Then, our common reduct is s_c: we have $a \twoheadrightarrow b \twoheadrightarrow s_b \twoheadrightarrow s_c$ and $a \twoheadrightarrow c \twoheadrightarrow s_c$. □

Surprisingly, the converse holds as well, as long as our ARS is countable – that is, there is an injective map $A \to \mathbb{N}$.

Lemma 5.2 ([4, p. 51]). *Any countable, confluent ARS $\mathcal{A} = (A, \to)$ has the cofinality property.*

```
 lemma cp_of_countable_cr [cnt: Countable α] (cr: confluent A.union_rel):
   cofinality_property A := ...
```

We have formalized the proof that is given in [4, p. 51].

Proof. Let $\mathcal{A} = (A, \to)$ be an ARS. Assume $\mathcal{A}$ is confluent. Assume A is countable.
Fix $a \in A$. We must show that there exists a reduction sequence $a = b_0 \to b_1 \to \cdots$ which is cofinal in $\mathcal{G}(a)$.
$\mathcal{G}(a)$ contains a, so it is nonempty. Since A is countable, there exists a function $f: \mathbb{N} \to \mathcal{G}(a)$ which is surjective. Consider f as a sequence. Every element in the sequence is a reduct of a, but they are not necessarily reducts of one another. However, we can use f to build a new sequence g:

$$\begin{aligned} g(0) &= a, \\ g(n+1) &= \text{the common reduct of } g(n) \text{ and } f(n). \end{aligned}$$

Every element in the codomain of both f and g is a reduct of a. Hence, a common reduct of $f(i)$ and $g(j)$ always exists, by confluence of $\mathcal{A}$.
Note that $g(n)$ forms a reduction sequence w.r.t. $\twoheadrightarrow$. That is,

$$g(0) \twoheadrightarrow g(1) \twoheadrightarrow g(2) \twoheadrightarrow \cdots$$

We can expand this reduction sequence into a reduction sequence w.r.t. $\to$,

$$g(0) \to \cdots \to g(1) \to \cdots \to g(2) \to \cdots$$

We wish to show that this sequence is cofinal in $\mathcal{G}(a)$. Assume $b \in \mathcal{G}(a)$. Since f is surjective, there must be an i s.t. $f(i) = b$. Then $g(i+1)$ is a reduct of b. □

5.1.1 Expansion of reduction sequences

The proof sketch above is relatively easy to formalize, save for one noteworthy detail: the step where g is expanded into a reduction sequence w.r.t. $\rightarrow$. This step is only mentioned very briefly in the original proof in [4], perhaps because Klop considered it obvious that such an expansion is possible.

For finite reduction sequences, we can easily prove this; our inductive definition is again well-suited to such a proof. It is easy to convince oneself that this also extends to the infinite case, but formalizing this turns out to be quite involved. We will look at the core parts of our Lean formalization, including some key definitions.

Our eventual goal is to prove that any reflexive-transitive reduction sequence $a_0 \twoheadrightarrow a_1 \twoheadrightarrow a_2 \twoheadrightarrow \cdots$ can be expanded to a reduction sequence $a_0 \rightarrow \cdots \rightarrow a_1 \rightarrow \cdots \rightarrow a_2 \rightarrow \cdots$. We will begin by proving a similar property for *transitive* reduction sequences, which is simpler because we have the guarantee that there are no ‘empty’ steps.

Lemma 5.3. *Any transitive step $a \rightarrow^+ b$ can be expanded into a list of elements $[a, \ldots, b]$ such that for any two adjacent elements x, y, we have $x \rightarrow y$.*

```
lemma trans_chain:
    r⁺ a b → ∃l, l.getLast? = some b ∧ List.Chain r a l := ...
```

Proof. By structural induction on the transitive step. □

Our formalized lemma makes use of `List.Chain`: a list `a::l` satisfying `List.Chain r a l` starts with a, and links each subsequent element via `r`.

In order to prove various other properties of these lists, we wish to have a function `trans_chain' {a b}: r⁺ a b → List α` which satisfies `(trans_chain' hstep).getLast? = some b` and `List.Chain r a (trans_chain' hstep)`, i.e. the property given by `trans_chain`. Unfortunately, we cannot compute such a list, but since we know one must exist, we can still define such a function as long as we mark it `noncomputable`. To do so, we use the function `Classical.choose`, which is derived from the type-theoretic axiom of choice.

Definition 5.2. If we have a reduction step $a \rightarrow^+ b$, we can get a list of elements $[\ldots, b]$ such that for any two adjacent elements x, y, we have $x \rightarrow y$.

```
noncomputable def trans_chain': r⁺ a b → List α :=
  fun h ↦ Classical.choose (trans_chain h)
```

Note that `trans_chain'` returns the list without the element a; this is convenient because we will want to concatenate multiple lists corresponding to a sequence $a \rightarrow^+ b \rightarrow^+ c \rightarrow^+ \ldots$, and the last element of the first step coincides with the first element of the second step, et cetera. In order to not contain these boundary elements twice, we always omit the first element of a step, and we prepend a later in the process.

The proof that `trans_chain'` satisfies the property of `l` in `trans_chain` is given by `Classical.choose_spec`, which has the type `(h: ∃x, p x) → p (Classical.choose h)`. Using this definition, we can for instance prove that such a list is always nonempty:

```
lemma trans_chain'.nonempty (h: r⁺ a b):
    trans_chain' h ≠ [] := ...
```

We can also use `trans_chain'` to define a sequence of lists satisfying the chain property.

Lemma 5.4. *Any transitive reduction sequence* $a_0 \to^+ a_1 \to^+ \cdots$ *can be transformed into a sequence of lists* $[[\dots, a_1], [\dots, a_2], \dots]$.

```
noncomputable def inf_trans_lists (f: ℕ → α) (hf: reduction_seq r⁺ ⊤ f): ℕ → List α
| n ⇒ trans_chain' (hf.inf_step n)
```

Proof. Trivial. □

We now define an auxiliary function `aux`, which allows us to get an element from such a sequence of lists using a pair of indices `(list_idx, elem_idx)`. Note that `elem_idx` may be larger than the list referred to by `list_idx`; in that case, the index 'rolls over' to the next list.

```
variable (l_seq: ℕ → List α) (hne: ∀n, (l_seq n) ≠ [])

noncomputable def aux (list_idx: ℕ) (elem_idx: ℕ) : α :=
  if h: elem_idx < (l_seq list_idx).length then
    (l_seq list_idx)[elem_idx]
  else
    have: elem_idx - (l_seq list_idx).length < elem_idx := by
      have := List.length_pos.mpr (hne list_idx)
      omega
    aux (list_idx + 1) (elem_idx - (l_seq list_idx).length)
```

This function is defined recursively; to prove that the function is terminating, we use the fact that each list in the sequence is nonempty (`hne`), and therefore `elem_idx` is decreasing. This is one reason we limit ourselves to transitive reduction sequences here – an equivalent definition for reflexive-transitive reduction sequences may have a potentially infinite amount of empty lists, making `aux` non-terminating.

If `hf: reduction_seq r⁺ ⊤ f`, then our desired expanded sequence is $f(0)$ followed by `aux (inf_trans_lists f hf) (inf_trans_lists.nonempty) 0 n`. We show this as follows.

Lemma 5.5. *The kth element starting from list* $m + 1$ *is reachable starting from list* m *by adding some* n *to* k *(namely, the length of list* m*).*

```
lemma aux_skip (m k: ℕ):
    ∃n, aux l_seq hne m (k + n) = aux l_seq hne (m + 1) k := ...
```

Proof. By definition of aux. □

Lemma 5.6. *We can get the kth element of list $m + i$ by getting the $k + n$th element of list m, where n is the length of the intermediate lists.*

```
lemma aux_skip_i (i m k: ℕ):
    ∃n, aux l_seq hne m (k + n) = aux l_seq hne (m + i) k := ...
```

Proof. By induction on i, along with Lemma 5.5. □

Lemma 5.7. *Let $a_0 \to^+ a_1 \to^+ \ldots$ be an infinite transitive reduction sequence. Each element a_n for $n > 0$ appears as an element in the sequence generated by* aux *using* `inf_trans_lists`*, starting at the $(n-1)$th list.*

```
lemma aux_elem' (f: ℕ → α) (hf: reduction_seq r⁺ ⊤ f) (n) (hn: n > 0):
    let ls := inf_trans_lists f hf
    f n = aux ls inf_trans_lists.nonempty (n - 1) ((ls (n - 1)).length - 1) := ...
```

Proof. By definition of `inf_trans_lists` and `trans_chain'`, the $(n-1)$th list is the list containing the elements $[\ldots, a_n]$. a_n is the last element of this list. □

Lemma 5.8. *Let $a_0 \to^+ a_1 \to^+ \ldots$ be an infinite transitive reduction sequence. Each element a_n for $n > 0$ appears in the sequence generated by* aux *using* `inf_trans_lists`*, starting at the first list.*

```
lemma aux_elem (f: ℕ → α) (hf: reduction_seq r⁺ ⊤ f) (n: ℕ) (hn: n > 0):
    ∃n', f n = aux (inf_trans_lists f hf) inf_trans_lists.nonempty 0 n' := ...
```

Proof. By Lemmas 5.6 and 5.7. □

Extending the expansion to reflexive-transitive reduction sequences

Definition 5.3 (Step guarantee)**.** Let $s_0 \twoheadrightarrow s_1 \twoheadrightarrow \cdots$ be an infinite reflexive-transitive reduction sequence. We say (s_n) satisfies the *weak step guarantee* if, for any index n, there exists an index $m \geq n$ such that $s_m \neq s_{m+1}$. That is, at any point in the sequence, there is a next step that is non-reflexive.

```
def weak_step_guarantee (f: ℕ → α)
  := ∀n, ∃m ≥ n, f m ≠ f (m + 1)
```

We say (s_n) satisfies the *step guarantee* if, for any index n, there exists an index $m \geq n$ such that $s_m \neq s_{m+1}$ and $\forall m', n < m' \leq m \rightarrow s_n = s'_m$.

```
def step_guarantee (f: ℕ → α)
  := ∀n, ∃m ≥ n, f m ≠ f (m + 1) ∧ (∀m' ∈ Set.Ioc n m, f n = f m')
```

Lemma 5.9. *If (s_n) satisfies the weak step guarantee, it satisfies the step guarantee.*

```
lemma sg_of_wsg (f: ℕ → α) (hwsg: weak_step_guarantee f):
    step_guarantee f := ...
```

Proof. Assume (s_n) satisfies the weak step guarantee, and let $n \in \mathbb{N}$. Let $S = \{m \mid m \geq n \wedge s_m \neq s_{m+1}\}$. By the weak step guarantee, S is nonempty. Since $(\mathbb{N}, <)$ is well-founded, there must be a minimal m such that $m \geq n \wedge s_m \neq s_{m+1}$, call it m^*.

We must show that, for all elements m' between n and m^*, $s_n = s'_m$. We proceed by contradiction. Let m' be between n and m^*, and assume $s_n \neq s'_m$. Since m^* is minimal, any m such that $n \leq m < m^*$ must have $s_m = s_{m+1}$. But then $s_n = s_{n+1} = \ldots = s'_m$, contradicting our assumption. □

Definition 5.4. Let (s_n) be an infinite reflexive-transitive reduction sequence which satisfies the step guarantee. Then we can define a derived sequence (s_n^+) as follows:

Lemma 5.10. *If an infinite reflexive-transitive reduction sequence (s_n) satisfies the step guarantee, there exists an infinite transitive reduction sequence (s'_n) which contains all elements of (s_n).*

5.2 Equivalence of componentwise definition

It turns out the componentwise and reduction-graph versions of the cofinality property are equivalent.

Lemma 5.11. *Let $\mathcal{A} = (A, \rightarrow)$ be an ARS. The regular and componentwise cofinality property are equivalent.*

Proof.

CP ⟹ CP$^{\equiv}$ Assume $\mathcal{A}$ is CP. Let $a \in A$, and assume (s_n) is a reduction sequence which is cofinal in the reduction graph of a. We wish to show that this sequence is also cofinal in the component of a.

Let b be an element in the component of a, that is, $a \equiv b$. By Lemma 5.1, $\mathcal{A}$ is confluent. By Lemma 3.4, $\mathcal{A}$ is also conversion confluent. Then, by conversion confluence, a and b have a common reduct, c. c is in the reduction graph of a, and so has a reduct in (s_n). Then b has a reduct in (s_n), and (s_n) is also cofinal in the component of a. Hence, $\mathcal{A}$ is CP$^{\equiv}$.

$\mathrm{CP}^{\equiv} \Longrightarrow \mathrm{CP}$ Assume $\mathcal{A}$ is $\mathrm{CP}^{\equiv}$. Let $a \in A$, and assume (s_n) is a reduction sequence which is cofinal in the component of a. Let b be in the reduction graph of a. Then, b is also in the component of a, as $\twoheadrightarrow \subseteq \equiv$. Then b has a reduct in (s_n), and $\mathcal{A}$ is CP. $\square$

We will use the component version of the cofinality property in our proofs of CP $\Rightarrow$ DCR and CP $\Rightarrow$ DCR_2, later.

6 Decreasing diagrams

As we have discussed in the introduction, we often want to know whether a reduction system is confluent, but unfortunately, confluence is an undecidable property. Instead of directly proving that a reduction system is confluent, one often shows that a system satisfies some *confluence criterion*, a set of properties which has been proven to imply confluence. We have already discussed one confluence criterion, namely Newman's lemma, in Section 4. In this section, we discuss another, namely that of *Decreasing Diagrams* [10].

Definition 6.1. For an ARS $\mathcal{A} = (A, \{\to_i \mid i \in I\})$ and a relation $< \subseteq I \times I$, we define

$$\to_{<i} = \bigcup_{j<i} \to_j$$

Additionally, we use $\to_{<i \cup <j}$ as shorthand for $\to_{<i} \cup \to_{<j}$.

In order to prove confluence by decreasing diagrams, we must label the steps in our rewriting system using a label set I that has a well-founded order $<$, and show that this labeled rewriting system is *locally decreasing* – that is, any two diverging steps $c \leftarrow_\alpha a \to_\beta b$ can always be joined again by reduction sequences as shown in Fig. 6.

If a rewriting system can be labeled in this way, we say it is *decreasing Church-Rosser* (DCR).

$$\begin{tikzcd}
a \arrow[rrr, "\alpha"] \arrow[ddd, "\beta"'] & & & b \arrow[d, two heads, dotted, "<\alpha"] \\
& & & \circ \arrow[d, dotted, "\beta", "="'] \\
& & & \circ \arrow[d, two heads, dotted, "<\alpha \cup <\beta"] \\
c \arrow[r, two heads, dotted, "<\beta"'] & \circ \arrow[r, dotted, "=", "\alpha"'] & \circ \arrow[r, two heads, dotted, "<\alpha \cup <\beta"'] & d
\end{tikzcd}$$

Figure 6: A decreasing elementary diagram.

It is shown in [10] that any rewriting system that is DCR is confluent. In essence, this is a strengthened version of weak confluence, which is on its own not strong enough to prove confluence, with the added strength coming from the restrictions on the labeling of the steps.

Decreasing diagrams is a *complete* method for proving confluence of countable systems. We might think that the power of decreasing diagrams lies in the potential complexity of the labeling – we can have as many labels as we like, with some complex well-founded order on them. However, this is not the case for countable systems: Endrullis et al. proved in [3] that 2-label DCR is a complete method for proving confluence of countable systems. In this section, we will discuss how we have formalized these completeness results in Lean.

6.1 Decreasing Diagrams in Lean

We begin by defining the union of reduction relations as in Definition 6.1.

```
/-- The union of reduction relations with an index smaller than i. -/
@[simp]
def ARS.union_lt [PartialOrder I] (A: ARS α I): I → Rel α α :=
  fun i x y ↦ ∃j, j < i ∧ A.rel j x y

/-- Enable the syntax `r ∪ s` for the union of two relations r, s. -/
instance Rel.instUnion: Union (Rel α β) where
  union := fun r₁ r₂ x y ↦ (r₁ x y) ∨ (r₂ x y)
```

We can then define the notion of a locally decreasing ARS as follows:

```
def locally_decreasing [PartialOrder I] [IsWellFounded I (· < ·)] (B: ARS α I) :=
  ∀a b c i j, B.rel i a b ∧ B.rel j a c →
    ∃d, ((B.union_lt i)∗ • (B.rel j)⁼ • (B.union_lt i ∪ B.union_lt j)∗) b d ∧
        ((B.union_lt j)∗ • (B.rel i)⁼ • (B.union_lt i ∪ B.union_lt j)∗) c d
```

The large center dots • represent relation composition ($x(R \bullet S)z \Leftrightarrow \exists y, xRySz$); these are used to keep the intermediate elements unnamed, just as in Fig. 6.

Finally, an ARS is DCR if there exists a *reduction-equivalent* ARS which is locally decreasing. This reduction-equivalent ARS is the 'labeled' version of our original ARS.

```
def DCR (A: ARS α I) :=
  ∃(J: Type) (_: PartialOrder J) (_: IsWellFounded J (· < ·)) (B: ARS α J),
    A.union_rel = B.union_rel ∧ locally_decreasing B
```

Although the definition of DCR is generic over the index type, we will generally use the natural numbers or a subset of them as indices, and use the standard less-than relation as our well-founded relation on ℕ. In order to represent the property of being n-label DCR, that is, having a reduction-equivalent, locally decreasing ARS which uses at most n labels, we separately define `DCRn`:

```
def DCRn (n: ℕ) (A: ARS α I) :=
  ∃(B: ARS α (Fin n)), A.union_rel = B.union_rel ∧ locally_decreasing B
```

We use the type containing exactly n inhabitants, `Fin n`, as our index type here – this type is isomorphic to $\{m \in \mathbb{N} \mid m < n\}$. It is trivial to prove that any ARS that is DCRn n for any n must be DCR.

6.2 Completeness for countable systems

Lemma 5.2 tells us that any countable, confluent system has the cofinality property. Then, in order to prove that DCR is complete for countable systems, it suffices to prove that any system that has the cofinality property is DCR.

Theorem 6.1 ([8, p. 766]). *Every ARS with the cofinality property is DCR.*

```
def cp_imp_dcr (hcp: cofinality_property A):
  DCR A := ...
```

Our formalization follows the proof in [8, p. 766]. For convenience, we reproduce a proof sketch below.

Definition 6.2 (Rewrite distance). Let $\mathcal{A} = (A, \rightarrow)$ be an ARS. The *rewrite distance* between an element $a \in A$ and one of its reducts $b \in \mathcal{G}(a)$, written $d(a, b)$, is the minimal length of a rewrite sequence between a and b.

The rewrite distance between an element a and a set of elements $X \subseteq A$ where $X \cap \mathcal{G}(a)$ is non-empty, written $d_X(a, X)$, is the minimal length of a rewrite sequence between a and some element $x \in X$, i.e. $d_X(a, X) = \min\{d(a, x) \mid x \in X \cap \mathcal{G}(a)\}$.

```
def is_reduction_seq_from (r: Rel α α) (a b: α) (f: ℕ → α) (N: ℕ) :=
  f 0 = a ∧ f N = b ∧ reduction_seq r N f

lemma exists_reduction_seq_in_set (X: Set α) (hX: ∃x ∈ X, r∗ a x):
    ∃N f x, x ∈ X ∧ is_reduction_seq_from r a x f N := ...

open Classical in
noncomputable def dX (a: α) (X: Set α) (hX: ∃x ∈ X, r∗ a x)
  := Nat.findX (exists_reduction_seq_in_set X hX)
```

Instead of basing d_X on d, we define it directly, choosing instead to define $d(a, b)$ as $d_X(a, \{b\})$. Before defining d_X, we first define the property of being a length-N reduction sequence from a to b (`is_reduction_seq_from`). Then, we prove that, as long as there is an element of X which is a reduct of a, there is some N such that there exists a reduction sequence from a to an element in X with length N.

Our definition of d_X uses the function `Nat.findX`, which, given a proof that there is some natural number satisfying a predicate `p`, returns the smallest natural number satisfying `p`. This is essentially a convenience function which makes use of the fact that $(\mathbb{N}, <)$ is well-founded, and therefore has a minimum on every non-empty set, in particular the set whose elements satisfy `p`. You will notice we need to mark this definition as noncomputable, as well as open the `Classical` namespace. If a predicate is decidable, `Nat.find` is able to compute the smallest natural number satisfying it – since this predicate is not, we need to mark the definition as noncomputable and open `Classical` to allow `Nat.find` to use `Classical.propDecidable`.

Proof sketch. Let $\mathcal{A} = (A, \rightarrow)$ be an ARS. Assume $\mathcal{A}$ has the cofinality property. Recall that CP is equivalent to $\mathrm{CP}^{\equiv}$ by Lemma 5.11.

To prove that $\mathcal{A}$ is DCR, it suffices to prove that every component of $\mathcal{A}$ is DCR.

Let $\mathcal{C}(a)$ be a component of $\mathcal{A}$. By definition, it contains a, and so is nonempty. By $\mathrm{CP}^{\equiv}$, it must contain a reduction sequence $s_0 \rightarrow s_1 \rightarrow \cdots$ such that every $b \in \mathcal{C}(a)$ reduces to some s_i. We may assume that (s_n) is acyclic, if not, we can modify it to make it acyclic while preserving cofinality.

We assign labels to steps in $\mathcal{C}(a)$ as follows:

(i) $b \rightarrow_0 c$ if $b \rightarrow c$ appears in (s_n), i.e. $b = s_i$ and $c = s_{i+1}$ for some i.

(ii) $b \rightarrow_{n+1} c$ if

□

6.2.1 Existence of acyclic main roads

If we require our main road to be acyclic, we need to prove that every component *has* an acyclic main road. We will prove that any cofinal reduction sequence that contains cycles can be modified to remove the cycles. In many ways, this argument is similar to the expansion of reduction sequences as addressed in Lemma 5.3: it is easy to convince oneself that any cofinal reduction sequence which contains cycles can be modified to remove the cycles, but a formal proof is somewhat involved; and the proof technique used is very similar.

Lemma 6.2. *Let (c_n) be a reduction sequence which is cofinal in $\mathcal{A} = (A, \rightarrow)$, and let c^* be an element which appears after all other elements in the sequence. Then c^* forms an acyclic reduction sequence which is cofinal in $\mathcal{A}$.*

Proof. Let $a \in A$. Since (c_n) is cofinal in $\mathcal{A}$, a reduces to some c_i. By assumption, c^* appears after c_i, so $c_i \twoheadrightarrow c^*$. Then, by transitivity, a reduces to c^*. □

Corollary 6.3. *Let (c_n) be a finite reduction sequence which is cofinal in $\mathcal{A} = (A, \rightarrow)$. Then its last element forms an acyclic reduction sequence which is cofinal in $\mathcal{A}$.*

Proof. By Lemma 6.2; the last element in (c_n) appears after all other elements in (c_n). □

Corollary 6.4. *Let (c_n) be an infinite reduction sequence which is cofinal in $\mathcal{A} = (A, \to)$, with an element c^* appearing infinitely often in (c_n). Then c^* forms an acyclic reduction sequence which is cofinal in $\mathcal{A}$.*

Proof. By Lemma 6.2; c^* appears infinitely often, so certainly after any other element in (c_n). □

Lemma 6.5. *Let (c_n) be an infinite reduction sequence which is cofinal in $\mathcal{A} = (A, \to)$, with all elements appearing finitely often in (c_n). Then there is an acyclic reduction sequence which is cofinal in $\mathcal{A}$.*

Proof. If all elements appear finitely often in (c_n), each element c_i must have a greatest index $g(i)$ at which it appears; that is, $g(i)$ is the greatest index such that $c_i = c_{g(i)}$. We can pick indices of (c_n) to construct an infinite sequence of indices (i_n):

$$\begin{aligned} i_0 &= g(0) \\ i_{n+1} &= g(i_n + 1) \end{aligned}$$

and define the sequence $c'_n = c_{i_n}$. We wish to prove that (c'_n) is an acyclic reduction sequence that is cofinal in $\mathcal{A}$.

To prove that (c'_n) is cofinal in $\mathcal{A}$, we must prove that any element in A reduces to some element in (c'_n). Obviously, $g(i) \geq i$. Then $i_{n+1} = g(i_n + 1) \geq i_n + 1 > i_n$, and (i_n) is strictly monotone. Since (i_n) is strictly monotone, (c'_n) contains elements at least as late as any element of (c_n). Then any element of (c_n) must reduce to some element of (c'_n), and by transitivity every element $a \in A$ must reduce to some element of (c'_n).

To show that (c'_n) is a reduction sequence, note that $c'_n = c_{i_n}$ and $c'_{n+1} = c_{i_{n+1}} = c_{g(i_n+1)} = c_{i_n+1}$ by definition. Since (c_n) is a reduction sequence, $c_{i_n} \to c_{i_n+1}$, and therefore $c'_n \to c'_{n+1}$.

Lastly, we wish to show that (c'_n) is acyclic, i.e. for all $n, m \in \mathbb{N}$, if $c'_n = c'_m$, then $n = m$. If $c'_n = c'_m$, then $c_{i_n} = c_{i_m}$. By definition, i_n and i_m are the greatest indices of their corresponding elements in (c_n). Then $i_n = i_m$, for otherwise one index would not be the greatest. By strict monotonicity of i, then, we must have $n = m$. □

Lemma 6.6. *If there is a reduction sequence which is cofinal in $\mathcal{A}$, there is an acyclic reduction sequence which is cofinal in $\mathcal{A}$.*

Proof. Immediate from Corollaries 6.3 and 6.4 and Lemma 6.5. □

References

[1] H.P. Barendregt. *The Lambda Calculus: its Syntax and Semantics*. Ed. by J. Barwise et al. Vol. 103. Studies in Logic and The Foundations of Mathematics. Elsevier, 1984.

[2] Nachum Dershowitz and Zohar Manna. "Proving termination with multiset orderings". In: *Commun. ACM* 22.8 (Aug. 1979), pp. 465–476. ISSN: 0001-0782. DOI: 10.1145/359138.359142. URL: https://doi.org/10.1145/359138.359142.

[3] Jörg Endrullis et al. "Decreasing Diagrams for Confluence and Commutation". In: *CoRR* abs/1901.10773 (2019). arXiv: 1901.10773. URL: http://arxiv.org/abs/1901.10773.

[4] J.W. Klop. "Combinatory Reduction Systems". PhD thesis. Rijksuniversiteit Utrecht, 1980. URL: https://eprints.illc.uva.nl/id/eprint/1876/1/HDS-33-Jan-Willem-Klop.text.pdf.

[5] Tobias Nipkow. *A new proof of the wellfoundednes of the multiset ordering*. Oct. 1998. URL: https://www.seas.upenn.edu/~sweirich/types/archive/1999-2003/msg00031.html (visited on 12/18/2024).

[6] Tobias Nipkow. *An Inductive Proof of the Wellfoundedness of the Multiset Order*. Oct. 1998. URL: https://www21.in.tum.de/~nipkow/Misc/multiset.ps (visited on 12/18/2024).

[7] Christian Sternagel and René Thiemann. *Abstract Rewriting*. 2010. URL: https://www.isa-afp.org/entries/Abstract-Rewriting.html (visited on 09/11/2024).

[8] Terese. *Term Rewriting Systems*. Cambridge University Press, 2003.

[9] René Thiemann and Christian Sternagel. "Certification of Termination Proofs Using CeTA". In: *Theorem Proving in Higher Order Logics*. Ed. by Stefan Berghofer et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 452–468. ISBN: 978-3-642-03359-9.

[10] Vincent Van Oostrom. "Confluence by Decreasing Diagrams". In: *Theoretical Computer Science* 126.2 (1994), pp. 259–280.

[11] Harald Zankl. *Confluence by Decreasing Diagrams – Formalized*. 2013. arXiv: 1210.1100 [cs.LO]. URL: https://arxiv.org/abs/1210.1100.